

```
Generame la entidad en base a esto CREATE TABLE
"A3TT"."TASK" ( "ID" NUMBER(19,0) GENERATED ALWAYS
AS IDENTITY MINVALUE 1 MAXVALUE
999999999999999999999999999999 INCREMENT BY 1 START WITH 1
CACHE 20 NOORDER NOCYCLE NOKEEP NOSCALE NOT
NULL ENABLE, "NAME" VARCHAR2(255 BYTE) DEFAULT
'Inserta Nombre', "DESCRIPTION" VARCHAR2(4000 BYTE)
DEFAULT 'Inserta Descripcion', "STATE" NUMBER(1,0)
DEFAULT 0 NOT NULL ENABLE, "USER_ID" NUMBER(19,0)
NOT NULL ENABLE, "CATEGORY_ID" NUMBER(19,0) NOT
NULL ENABLE, CONSTRAINT "TASK_PK" PRIMARY KEY
("ID") USING INDEX PCTFREE 10 INITRANS 2 MAXTRANS 255
COMPUTE STATISTICS TABLESPACE "USERS" ENABLE,
CONSTRAINT "CATEGORY_FK" FOREIGN KEY
("CATEGORY_ID") REFERENCES "A3TT"."CATEGORY" ("ID")
ON DELETE CASCADE ENABLE, CONSTRAINT "USERS_FK"
FOREIGN KEY ("USER_ID") REFERENCES "A3TT"."USERS"
("ID") ON DELETE CASCADE ENABLE ) SEGMENT CREATION
DEFERRED PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS
255 NOCOMPRESS LOGGING TABLESPACE "USERS" ;
COMMENT ON COLUMN "A3TT"."TASK"."ID" IS 'Clave
primaria'; COMMENT ON COLUMN "A3TT"."TASK"."STATE"
IS '0 -> pendiente, 1 -> completada'; Entidad de ejemplo: import {
Users } from '@users/entities/users.entity'; import { Entity,
PrimaryGeneratedColumn, Column, ManyToOne, JoinColumn,
PrimaryColumn } from 'typeorm'; @Entity({ name: 'CATEGORY' })
export class Category { @PrimaryColumn({ name: 'ID', type:
'number', precision: 19, scale: 0, generated: 'identity', }) id: number;
@Column({ name: 'NAME', type: 'varchar2', length: 255, nullable:
false }) name: string; @ManyToOne(() => Users, { onDelete:
'CASCADE', nullable: false}) // ¡No hay (user) => user.categories!
@JoinColumn({ name: 'USER_ID', foreignKeyName:
'USER_ID' }) // Columna FK en BD user: Users; // Accedes al usuario
con `category.user` (opcional) }
```

## Entidad Task basada en la tabla TASK

```
typescript
import { Entity, PrimaryGeneratedColumn, Column, ManyToOne, JoinColumn
} from 'typeorm';
import { Users } from '@users/entities/users.entity';
import { Category } from '@category/entities/category.entity';

@Entity({ name: 'TASK', schema: 'A3TT' })
export class Task {
```

```

@PrimaryGeneratedColumn({
  name: 'ID',
  type: 'number',
  generated: 'identity',
  precision: 19,
  scale: 0,
  comment: 'Clave primaria'
})
id: number;

@Column({
  name: 'NAME',
  type: 'varchar2',
  length: 255,
  default: 'Inserta Nombre',
  nullable: true
})
name: string;

@Column({
  name: 'DESCRIPTION',
  type: 'varchar2',
  length: 4000,
  default: 'Inserta Descripcion',
  nullable: true
})
description: string;

@Column({
  name: 'STATE',
  type: 'number',
  precision: 1,
  scale: 0,
  default: 0,
  nullable: false,
  comment: '0 -> pendiente, 1 -> completada'
})
state: number;

@ManyToOne(() => Users, { onDelete: 'CASCADE', nullable: false })
@JoinColumn({
  name: 'USER_ID',
  referencedColumnName: 'id',
  foreignKeyName: 'USERS_FK'
})
user: Users;

@ManyToOne(() => Category, { onDelete: 'CASCADE', nullable: false })
@JoinColumn({
  name: 'CATEGORY_ID',
  referencedColumnName: 'id',
  foreignKeyName: 'CATEGORY_FK'
})
category: Category;
}

```

## Explicación de la entidad:

1. **Decorador @Entity:** Se especifica el nombre de la tabla ('TASK') y el esquema ('A3TT').

2. **ID:**
  - Es la clave primaria generada automáticamente como identidad.
  - Se mapea exactamente con la definición de la tabla (NUMBER(19,0)).
3. **Columnas NAME y DESCRIPTION:**
  - Tienen valores por defecto como en la tabla.
  - Se especifica su tipo (varchar2) y longitud máxima.
4. **Columna STATE:**
  - Tiene valor por defecto 0 y no puede ser nula.
  - Se añade el comentario que explica los valores posibles.
5. **Relaciones:**
  - **user:** Relación ManyToOne con Users, con eliminación en cascada.
  - **category:** Relación ManyToOne con Category, con eliminación en cascada.
  - Ambas relaciones usan @JoinColumn para especificar el nombre de la columna FK y el nombre de la constraint.

La entidad refleja fielmente la estructura de la tabla de base de datos, incluyendo todas las constraints y propiedades definidas.

```
Ahira Creame la migración export class CreateTaskTable1752919015934 implements MigrationInterface { Migración de ejemplo: import { MigrationInterface, QueryRunner } from "typeorm"; export class CreateTable implements MigrationInterface { name = 'CreateCategoryTable' public async up(queryRunner: QueryRunner): Promise<void> { await queryRunner.query(` CREATE TABLE "A3TT"."CATEGORY" ( "ID" NUMBER(19,0) GENERATED ALWAYS AS IDENTITY MINVALUE 1 MAXVALUE 999999999999999999999999 INCREMENT BY 1 START WITH 1 CACHE 20 NOORDER NOCYCLE NOKEEP NOSCALE NOT NULL ENABLE, "NAME" VARCHAR2(255 BYTE) NOT NULL ENABLE, "USER_ID" NUMBER(19,0) NOT NULL ENABLE, CONSTRAINT "CATEGORY_PK" PRIMARY KEY ("ID") USING INDEX PCTFREE 10 INITRANS 2 MAXTRANS 255 COMPUTE STATISTICS TABLESPACE "USERS" ENABLE, CONSTRAINT "USER_ID" FOREIGN KEY ("USER_ID") REFERENCES "A3TT"."USERS" ("ID") ON DELETE CASCADE ENABLE ) SEGMENT CREATION DEFERRED PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 NOCOMPRESS LOGGING TABLESPACE "USERS" `); await queryRunner.query(` COMMENT ON TABLE "A3TT"."CATEGORY" IS 'Tabla de categorías del sistema' `); await queryRunner.query(` COMMENT ON COLUMN "A3TT"."CATEGORY"."ID" IS 'Clave primaria' `); await queryRunner.query(` COMMENT ON COLUMN "A3TT"."CATEGORY"."NAME" IS 'Nombre de la categoría' `); } public async down(queryRunner: QueryRunner): Promise<void> { await queryRunner.query(` DROP TABLE "A3TT"."CATEGORY" CASCADE CONSTRAINTS`); }}
```

## Migración para la tabla TASK

```
typescript
import { MigrationInterface, QueryRunner } from "typeorm";
```



- Foreign Keys (CATEGORY\_FK y USERS\_FK) con ON DELETE CASCADE
- 3. **Configuración de almacenamiento:**
  - Tablespace USERS
  - Parámetros PCTFREE, PCTUSED, INITRANS, MAXTRANS
  - Opciones de NOCOMPRESS y LOGGING
- 4. **Comentarios:**
  - Se incluyen los comentarios para las columnas ID y STATE como en la definición original
- 5. **Método down:**
  - Elimina la tabla con CASCADE CONSTRAINTS para asegurar que se eliminan también las constraints

Esta migración replica exactamente la estructura de la tabla TASK como fue definida en el SQL original, incluyendo todos los detalles técnicos de la configuración de Oracle.

**Generame CreateUpdateTaskDto. Este debe tener nombre y descripción únicamente export class CreateUpdateTaskDto {}**

## CreateUpdateTaskDto para la entidad Task

```
typescript
import { IsString, IsOptional, MaxLength, MinLength } from 'class-validator';

export class CreateUpdateTaskDto {
  @IsString({ message: 'El nombre debe ser una cadena de texto' })
  @MinLength(3, { message: 'El nombre debe tener al menos 3 caracteres' })
  @MaxLength(255, { message: 'El nombre no puede exceder los 255 caracteres' })
  @IsOptional()
  name?: string;

  @IsString({ message: 'La descripción debe ser una cadena de texto' })
  @MinLength(3, { message: 'La descripción debe tener al menos 3 caracteres' })
  @MaxLength(4000, { message: 'La descripción no puede exceder los 4000 caracteres' })
  @IsOptional()
  description?: string;
}
```

### Características del DTO:

1. **Validaciones:**
  - @IsString(): Valida que ambos campos sean strings
  - @MinLength(3): Requiere mínimo 3 caracteres

- `@MaxLength()`: Limita a los máximos definidos en la entidad (255 para name, 4000 para description)
  - `@IsOptional()`: Hace que ambos campos sean opcionales para permitir actualizaciones parciales
2. **Tipado:**
- Ambos campos son opcionales (usando `?`) para permitir su uso tanto en creación como actualización
  - Se mantienen los mismos nombres de campos que en la entidad
3. **Mensajes personalizados:**
- Cada decorador incluye mensajes de error claros para una mejor experiencia de desarrollo

Este DTO es ideal para usar tanto en endpoints de creación (POST) como de actualización (PATCH o PUT) de tareas, ya que todos los campos son opcionales pero con validaciones cuando están presentes.

import { ApiProperty } from '@nestjs/swagger'; import { IsString, IsNotEmpty } from 'class-validator'; export class CreateUpdateCategoryDto { @ApiProperty({ description: 'Nombre del usuario' }) @IsString() @IsNotEmpty() name: string; } Entonces este se podría mejorar?

## Improved CreateUpdateCategoryDto with Swagger and Validation

Here's an enhanced version of your DTO with better documentation and validation:

```
typescript
import { ApiProperty } from '@nestjs/swagger';
import { IsString, IsNotEmpty, Length, Matches } from 'class-validator';

export class CreateUpdateCategoryDto {
  @ApiProperty({
    description: 'Nombre de la categoría',
    example: 'Tareas personales',
    minLength: 3,
    maxLength: 255,
    pattern: '^[a-zA-ZáéíóúÁÉÍÓÚñÑ\\s]+$'
  })
  @IsString({ message: 'El nombre debe ser una cadena de texto válida' })
  @IsNotEmpty({ message: 'El nombre no puede estar vacío' })
  @Length(3, 255, {
    message: 'El nombre debe tener entre 3 y 255 caracteres'
  })
  @Matches(/^([a-zA-ZáéíóúÁÉÍÓÚñÑ\\s])+$/, {
    message: 'El nombre solo puede contener letras y espacios'
  })
  name: string;
}
```

### Key Improvements:

1. **Enhanced Swagger Documentation:**
  - Added example value for better API documentation
  - Included minLength, maxLength and pattern in @ApiProperty
  - More descriptive property description
2. **Better Validation:**
  - Added @Length decorator to enforce size limits (matches your entity definition)
  - Added @Matches with regex pattern to:
    - Allow only letters and spaces
    - Support Spanish characters (áéíóúñ)
  - Custom error messages for each validation case
3. **Consistency:**
  - Matches your database column definition (VARCHAR2(255))
  - Follows REST API best practices for DTOs
4. **Security:**
  - The regex pattern helps prevent potential injection attacks by restricting input characters

For even more robust validation, you could consider adding:

```
typescript
import { Transform } from 'class-transformer';

// Add this above the name property
@Transform(({ value }) => value?.trim())
```

This would automatically trim whitespace from the input value. Would you like me to include that as well?

**Creo que lo del pattern es innecesario, mejor que el usuario ponga lo que quiera**

## Simplified and Flexible CreateUpdateCategoryDto

Entiendo tu punto. Aquí está una versión simplificada que mantiene la validación básica pero da más libertad al usuario:

```
typescript
import { ApiProperty } from '@nestjs/swagger';
import { IsString, IsNotEmpty, Length } from 'class-validator';

export class CreateUpdateCategoryDto {
  @ApiProperty({
    description: 'Nombre de la categoría',
    examples: {
      'Ejemplo básico': { value: 'Tareas personales' },
      'Ejemplo con números': { value: 'Proyecto 2023' },
      'Ejemplo con símbolos': { value: '¡Urgente!' }
    },
  })
  name: string;
```

```

    minLength: 3,
    maxLength: 255
  })
  @IsString({ message: 'El nombre debe ser un texto válido' })
  @NotEmpty({ message: 'El nombre es requerido' })
  @Length(3, 255, {
    message: 'El nombre debe tener entre 3 y 255 caracteres'
  })
  name: string;
}

```

## Mejoras realizadas:

- Eliminada la restricción de pattern:**
  - Ahora acepta cualquier caracter que cumpla con ser string
  - Permite números, símbolos y cualquier caracter especial
- Mejor documentación Swagger:**
  - Múltiples ejemplos que muestran la flexibilidad permitida
  - Ejemplos con diferentes casos de uso (símbolos, números)
- Validación básica mantenida:**
  - Longitud mínima/máxima (3-255 caracteres)
  - Campo requerido
  - Tipo string
- Mensajes de error claros:**
  - Simple y directos al punto

Esta versión es ideal si:

- Quieres dar máxima flexibilidad a los usuarios
- Prefieres sanitizar/limpiar los datos en otro lugar (service)
- Tu aplicación necesita aceptar diversos formatos de nombres

¿Te gustaría que agregue alguna otra característica o prefieres mantenerlo así de simple?

```

import { ApiProperty } from '@nestjs/swagger'; import { IsEmail, IsNotEmpty,
IsString, MinLength } from 'class-validator'; export class LoginUserDto {
  @ApiProperty({ description: 'Nombre del usuario' }) @IsEmail() email: string;
  @ApiProperty({ description: 'Nombre del usuario' }) @IsString() @MinLength(8)
password: string; } import { ApiProperty } from '@nestjs/swagger'; import {
IsEmail, IsString, MinLength, MaxLength } from 'class-validator'; export class
CreateUserDto { @ApiProperty({ description: 'Nombre del usuario' }) @IsString()
@MinLength(3) @MaxLength(20) name: string; @ApiProperty({ description:
'Nombre del usuario' }) @IsEmail() email: string; @ApiProperty({ description:
'Nombre del usuario' }) @IsString() @MinLength(8) password: string; } import {
ApiProperty, ApiPropertyOptional } from '@nestjs/swagger'; import { IsEmail,
IsString, MinLength, MaxLength, IsOptional } from 'class-validator'; export class
UpdateUserDto { @ApiPropertyOptional({ description: 'Nombre del usuario' })
@IsOptional() @IsString() @MinLength(3) @MaxLength(20) name: string;
@ApiPropertyOptional({ description: 'Nombre del usuario' }) @IsOptional()
@IsString() @MinLength(8) password: string; } ¿Y estos de aquí son
mejorables???

```



# Improved User DTOs with Better Documentation and Validation

Here are enhanced versions of your user-related DTOs with better documentation, validation, and consistency:

## 1. LoginUserDto (Improved)

typescript

```
import { ApiProperty } from '@nestjs/swagger';
import { IsEmail, IsNotEmpty, IsString, MinLength, MaxLength } from
'class-validator';

export class LoginUserDto {
  @ApiProperty({
    description: 'Email del usuario',
    example: 'usuario@ejemplo.com',
    maxLength: 100
  })
  @IsEmail({}, { message: 'Debe proporcionar un email válido' })
  @IsNotEmpty({ message: 'El email es requerido' })
  @MaxLength(100, { message: 'El email no puede exceder los 100
caracteres' })
  email: string;

  @ApiProperty({
    description: 'Contraseña del usuario',
    example: 'MiContraseñaSegural23',
    minLength: 8,
    maxLength: 64
  })
  @IsString({ message: 'La contraseña debe ser una cadena de texto'
})
  @IsNotEmpty({ message: 'La contraseña es requerida' })
  @MinLength(8, { message: 'La contraseña debe tener al menos 8
caracteres' })
  @MaxLength(64, { message: 'La contraseña no puede exceder los 64
caracteres' })
  password: string;
}
```

## 2. CreateUserDto (Improved)

typescript

```
import { ApiProperty } from '@nestjs/swagger';
import { IsEmail, IsString, MinLength, MaxLength, Matches, IsNotEmpty
} from 'class-validator';

export class CreateUserDto {
  @ApiProperty({
    description: 'Nombre completo del usuario',
    example: 'Juan Pérez',
    minLength: 3,
    maxLength: 50
  })
}
```

```

    @IsString({ message: 'El nombre debe ser una cadena de texto' })
    @NotEmpty({ message: 'El nombre es requerido' })
    @MinLength(3, { message: 'El nombre debe tener al menos 3
caracteres' })
    @MaxLength(50, { message: 'El nombre no puede exceder los 50
caracteres' })
    @Matches(/^[a-zA-ZáéíóúÁÉÍÓÚñÑ\s]+$/, {
        message: 'El nombre solo puede contener letras y espacios'
    })
    name: string;

    @ApiProperty({
        description: 'Email del usuario',
        example: 'usuario@ejemplo.com',
        maxLength: 100
    })
    @IsEmail({}, { message: 'Debe proporcionar un email válido' })
    @NotEmpty({ message: 'El email es requerido' })
    @MaxLength(100, { message: 'El email no puede exceder los 100
caracteres' })
    email: string;

    @ApiProperty({
        description: 'Contraseña del usuario (mínimo 8 caracteres, 1
mayúscula, 1 número)',
        example: 'MiContraseña123',
        minLength: 8,
        maxLength: 64
    })
    @IsString({ message: 'La contraseña debe ser una cadena de texto'
    })
    @NotEmpty({ message: 'La contraseña es requerida' })
    @MinLength(8, { message: 'La contraseña debe tener al menos 8
caracteres' })
    @MaxLength(64, { message: 'La contraseña no puede exceder los 64
caracteres' })
    @Matches(/(?=.*[A-Z])(?=.*[0-9]).*/, {
        message: 'La contraseña debe contener al menos una mayúscula y
un número'
    })
    password: string;
}

```

### 3. UpdateUserDto (Improved)

typescript

```

import { ApiPropertyOptional } from '@nestjs/swagger';
import { IsString, MinLength, MaxLength, Matches, IsOptional } from
'class-validator';

export class UpdateUserDto {
    @ApiPropertyOptional({
        description: 'Nuevo nombre del usuario',
        example: 'María García',
        minLength: 3,
        maxLength: 50
    })
    @IsOptional()
    @IsString({ message: 'El nombre debe ser una cadena de texto' })

```

```

        @MinLength(3, { message: 'El nombre debe tener al menos 3
caracteres' })
        @MaxLength(50, { message: 'El nombre no puede exceder los 50
caracteres' })
        @Matches(/^([a-zA-ZáéíóúÁÉÍÓÚñÑ\s])+$/, {
            message: 'El nombre solo puede contener letras y espacios'
        })
        name?: string;

        @ApiPropertyOptional({
            description: 'Nueva contraseña (mínimo 8 caracteres, 1
mayúscula, 1 número)',
            example: 'NuevaContraseña456',
            minLength: 8,
            maxLength: 64
        })
        @IsOptional()
        @IsString({ message: 'La contraseña debe ser una cadena de texto'
})
        @MinLength(8, { message: 'La contraseña debe tener al menos 8
caracteres' })
        @MaxLength(64, { message: 'La contraseña no puede exceder los 64
caracteres' })
        @Matches(/(?=.*[A-Z])(?=.*[0-9]).*/, {
            message: 'La contraseña debe contener al menos una mayúscula y
un número'
        })
        password?: string;
    }

```

## Key Improvements:

1. **Better Documentation:**
  - More descriptive property descriptions
  - Clear examples for each field
  - Added length constraints in documentation
2. **Enhanced Validation:**
  - Added maximum length validations
  - Password strength requirements (uppercase + number)
  - Name format validation (letters and spaces only)
  - Clear error messages
3. **Consistency:**
  - Uniform length constraints across DTOs
  - Consistent validation patterns
  - Better property naming in documentation
4. **Security:**
  - Stronger password requirements
  - Input sanitization through validation
5. **Optional Fields:**
  - Clear distinction between required and optional fields
  - Better nullability handling in UpdateUserDto

Would you like me to adjust any specific aspect of these improvements?

Seguro que es lo mejor validar en los DTOs y entonces las validaciones y documentaciones de controladores serían a parte, para los endpoints? Explicame esto

# Validación en DTOs vs Controladores: Arquitectura óptima

Efectivamente, hay una separación clara de responsabilidades entre DTOs y controladores en NestJS. Te explico el enfoque recomendado:

## 1. Validación en DTOs (Capa de Presentación)

**Propósito:** Validar la estructura y formato de los datos de entrada

**Ventajas:**

- **Validación declarativa:** Usas decoradores para definir reglas
- **Reutilización:** El mismo DTO puede usarse en múltiples endpoints
- **Documentación automática:** Swagger genera documentación basada en los decoradores
- **Separación de preocupaciones:** El controlador no maneja validación básica

**Ejemplo (DTO):**

```
typescript
export class CreateUserDto {
  @ApiProperty({ description: 'Nombre del usuario' })
  @IsString()
  @MinLength(3)
  @MaxLength(20)
  name: string;

  // ... otras propiedades
}
```

## 2. Validación en Controladores (Lógica de Negocio)

**Propósito:** Validar reglas de negocio más complejas

**Cuándo usarlo:**

- Verificar existencia de recursos relacionados
- Validar permisos/autorizaciones
- Lógica condicional compleja
- Reglas que requieren acceso a la base de datos

**Ejemplo (Controller):**

```
typescript
```

```

@Post()
async create(@Body() createUserDto: CreateUserDto) {
  // Validación de negocio (ej: email único)
  const userExists = await
this.userService.findByEmail(createUserDto.email);
  if (userExists) {
    throw new ConflictException('El email ya está registrado');
  }

  return this.userService.create(createUserDto);
}

```

### 3. Documentación en Controladores

#### Mejor práctica:

- La documentación básica va en los DTOs (como viste)
- La documentación del endpoint (responses, códigos de estado) va en el controlador

#### Ejemplo completo:

```

typescript
@ApiTags('Users')
@Controller('users')
export class UsersController {
  constructor(private readonly userService: UsersService) {}

  @Post()
  @ApiOperation({ summary: 'Crear nuevo usuario' })
  @ApiResponse({
    status: 201,
    description: 'Usuario creado exitosamente',
    type: UserResponseDto
  })
  @ApiResponse({
    status: 409,
    description: 'El email ya está registrado'
  })
  async create(@Body() createUserDto: CreateUserDto) {
    // Validación de negocio aquí
  }
}

```

### ¿Por qué esta separación?

1. **Single Responsibility Principle:** Cada componente tiene una sola responsabilidad
2. **Mantenibilidad:** Cambios en reglas de validación no afectan la lógica del controlador
3. **Claridad:** El código es más fácil de entender y depurar
4. **Testing:** Puedes testear validaciones y lógica de negocio por separado

### Flujo típico de una petición:

1. **Validación de formato:** Automática (DTO + ValidationPipe)
2. **Validación de negocio:** Manual (en el controlador o servicio)
3. **Procesamiento:** Si pasa todas las validaciones

## Conclusión:

Sí, es la mejor práctica validar lo básico en DTOs y dejar las validaciones complejas para los controladores/servicios. Esto crea una arquitectura limpia y mantenible.

entonces los filters sería una alternativa a todo esto, o como funciona?

# Filters vs DTOs + Controllers: Complementos en la arquitectura NestJS

Los **filters** (filtros de excepciones) no son una alternativa a la validación con DTOs y controladores, sino un **mecanismo complementario** para manejo de errores. Te explico cómo funcionan juntos:

## 1. Diferencia fundamental

DTOs + Validación	Filters
Validación <i>preventiva</i> de datos	Manejo <i>reactivo</i> de errores
Evita que datos inválidos lleguen a la lógica	Captura y formatea errores que ya ocurrieron
Decoradores ( <code>@IsEmail()</code> , etc.)	Clases con <code>@Catch()</code>

## 2. Cómo funcionan los filters

Los filters interceptan excepciones no capturadas para:

typescript

```
import { ExceptionFilter, Catch, ArgumentsHost } from
'@nestjs/common';

@Catch(ConflictException)
export class ConflictExceptionFilter implements ExceptionFilter {
  catch(exception: ConflictException, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse();

    response.status(409).json({
      statusCode: 409,
      timestamp: new Date().toISOString(),
      message: 'Recurso duplicado: ' + exception.message,
    });
  }
}
```

### 3. Cuándo usar filters (ejemplos prácticos)

#### 1. Formateo consistente de errores:

typescript

```
• @UseFilters(new HttpExceptionFilter())
@Post()
async create(@Body() dto: CreateUserDto) {
  // ...
}
```

#### • Manejo de errores específicos:

typescript

```
2. @Catch(MongoServerError)
3. export class MongoErrorFilter implements ExceptionFilter {
4.   catch(exception: MongoServerError, host: ArgumentsHost) {
5.     if (exception.code === 11000) {
6.       throw new ConflictException('Registro duplicado');
7.     }
8.   }
9. }
```

### 4. Flujo completo de una petición con ambos

1. **Validación con DTO** (automática via `ValidationPipe`):
  - Si falla → Lanza `BadRequestException`
2. **Validación en controlador**:
  - Si falla → Lanza excepciones personalizadas (ej: `ConflictException`)
3. **Filter intercepta**:
  - Captura la excepción
  - Decide cómo formatear la respuesta
  - Loggea el error si es necesario

### 5. Configuración recomendada

**Globalmente** (en `main.ts`):

typescript

```
app.useGlobalPipes(
  new ValidationPipe({
    whitelist: true,
    forbidNonWhitelisted: true
  })
);

app.useGlobalFilters(
  new AllExceptionsFilter(),
  new HttpExceptionFilter()
);
```

**Localmente** (para casos específicos):

```
typescript
@UseFilters(new DatabaseExceptionFilter())
@Controller('users')
export class UsersController {
  // ...
}
```

## 6. ¿Por qué no reemplazan la validación en DTOs?

1. Los filters no previenen errores, solo los manejan
2. No proporcionan documentación automática (Swagger)
3. No validan estructura de datos, solo reaccionan a errores

## Conclusión

Usa **ambos** en conjunto:

- **DTOs + ValidationPipe:** Para validación de estructura de datos
- **Validación en controladores:** Para reglas de negocio
- **Filters:** Para manejo consistente de errores y respuestas

Esta combinación sigue el principio de **separación de preocupaciones** y crea una API robusta y mantenible.

al final lo he dejado así para que sea más consistente con la capacidad de la BBDD

```
import { ApiProperty, ApiPropertyOptional } from '@nestjs/swagger'; import { IsEmail, IsString, MinLength, MaxLength, IsOptional, Matches } from 'class-validator'; export class UpdateUserDto { @ApiPropertyOptional({ description: 'Nuevo nombre del usuario', example: 'María García', minLength: 3, maxLength: 50 }) @IsOptional() @IsString({ message: 'El nombre debe ser una cadena de texto' }) @MinLength(3, { message: 'El nombre debe tener al menos 3 caracteres' }) @MaxLength(30, { message: 'El nombre no puede exceder los 50 caracteres' }) name?: string; @ApiPropertyOptional({ description: 'Nueva contraseña (mínimo 8 caracteres, 1 mayúscula, 1 número)', example: 'NuevaContraseña456', minLength: 8, maxLength: 64 }) @IsOptional() @IsString({ message: 'La contraseña debe ser una cadena de texto' }) @MinLength(8, { message: 'La contraseña debe tener al menos 8 caracteres' }) @MaxLength(255, { message: 'La contraseña no puede exceder los 255 caracteres' })
```



```
@Matches(/(?=.*[A-Z])(?=.*[0-9]).*/, { message: 'La
contraseña debe contener al menos una mayúscula y un
número' }) password?: string; import { ApiProperty } from
'@nestjs/swagger'; import { IsEmail, IsNotEmpty, IsString,
MaxLength, MinLength } from 'class-validator'; export class
LoginUserDto { @ApiProperty({ description: 'Email del
usuario', example: 'usuario@ejemplo.com', maxLength: 255
}) @IsEmail({}, { message: 'Debe proporcionar un email
válido' }) @IsNotEmpty({ message: 'El email es requerido' })
@MaxLength(255, { message: 'El email no puede exceder los
255 caracteres' }) email: string; @ApiProperty({ description:
'Contraseña del usuario', example:
'MiContraseñaSegura123', minLength: 8, maxLength: 255 })
@IsString({ message: 'La contraseña debe ser una cadena de
texto' }) @IsNotEmpty({ message: 'La contraseña es
requerida' }) @MinLength(8, { message: 'La contraseña debe
tener al menos 8 caracteres' }) @MaxLength(255, { message:
'La contraseña no puede exceder los 255 caracteres' })
password: string; } import { ApiProperty } from
'@nestjs/swagger'; import { IsEmail, IsString, MinLength,
MaxLength, Matches, IsNotEmpty } from 'class-validator';
export class CreateUserDto { @ApiProperty({ description:
'Nombre completo del usuario', example: 'Juan Pérez',
minLength: 3, maxLength: 30}) @IsString({ message: 'El
nombre debe ser una cadena de texto' }) @IsNotEmpty({
message: 'El nombre es requerido' }) @MinLength(3, {
message: 'El nombre debe tener al menos 3 caracteres' })
@MaxLength(30, { message: 'El nombre no puede exceder los
30 caracteres' }) name: string; @ApiProperty({ description:
'Email del usuario', example: 'usuario@ejemplo.com',
maxLength: 255 }) @IsEmail({}, { message: 'Debe
proporcionar un email válido' }) @IsNotEmpty({ message: 'El
email es requerido' }) @MaxLength(255, { message: 'El email
no puede exceder los 255 caracteres' }) email: string;
@ApiProperty({ description: 'Contraseña del usuario
(mínimo 8 caracteres, 1 mayúscula, 1 número)', example:
'MiContraseña123', minLength: 8, maxLength: 255 })
@IsString({ message: 'La contraseña debe ser una cadena de
```

```
texto' }) @IsEmpty({ message: 'La contraseña es
requerida' }) @MinLength(8, { message: 'La contraseña debe
tener al menos 8 caracteres' }) @MaxLength(255, { message:
'La contraseña no puede exceder los 255 caracteres' })
@Matches(/(?=.*[A-Z])(?=.*[0-9]).*/, { message: 'La
contraseña debe contener al menos una mayúscula y un
número' }) password: string; }
```

## Revisión y Mejora Final de los DTOs de Usuario

Veo que has hecho un excelente trabajo ajustando los DTOs para que sean consistentes con las capacidades de tu base de datos. Aquí está mi revisión detallada con algunas sugerencias finales:

### 1. UpdateUserDto (Mejora Final)

```
typescript
import { ApiPropertyOptional } from '@nestjs/swagger';
import { IsString, MinLength, MaxLength, IsOptional, Matches } from
'class-validator';

export class UpdateUserDto {
  @ApiPropertyOptional({
    description: 'Nuevo nombre del usuario',
    example: 'María García',
    minLength: 3,
    maxLength: 30
  })
  @IsOptional()
  @IsString({ message: 'El nombre debe ser una cadena de texto' })
  @MinLength(3, { message: 'El nombre debe tener al menos 3
caracteres' })
  @MaxLength(30, { message: 'El nombre no puede exceder los 30
caracteres' })
  name?: string;

  @ApiPropertyOptional({
    description: 'Nueva contraseña (mínimo 8 caracteres, 1
mayúscula, 1 número)',
    example: 'NuevaContraseña456',
    minLength: 8,
    maxLength: 255
  })
  @IsOptional()
  @IsString({ message: 'La contraseña debe ser una cadena de texto'
})
  @MinLength(8, { message: 'La contraseña debe tener al menos 8
caracteres' })
  @MaxLength(255, { message: 'La contraseña no puede exceder los 255
caracteres' })
```

```

    @Matches(/(?!.*[A-Z])(?!.*[0-9]).*/, {
      message: 'La contraseña debe contener al menos una mayúscula y
un número'
    })
    password?: string;
  }

```

## 2. LoginUserDto (Mejora Final)

typescript

```

import { ApiProperty } from '@nestjs/swagger';
import { IsEmail, IsNotEmpty, IsString, MaxLength, MinLength } from
'class-validator';

export class LoginUserDto {
  @ApiProperty({
    description: 'Email del usuario',
    example: 'usuario@ejemplo.com',
    maxLength: 255
  })
  @IsEmail({}, { message: 'Debe proporcionar un email válido' })
  @IsNotEmpty({ message: 'El email es requerido' })
  @MaxLength(255, { message: 'El email no puede exceder los 255
caracteres' })
  email: string;

  @ApiProperty({
    description: 'Contraseña del usuario',
    example: 'MiContraseñaSegura123',
    minLength: 8,
    maxLength: 255
  })
  @IsString({ message: 'La contraseña debe ser una cadena de texto'
})
  @IsNotEmpty({ message: 'La contraseña es requerida' })
  @MinLength(8, { message: 'La contraseña debe tener al menos 8
caracteres' })
  @MaxLength(255, { message: 'La contraseña no puede exceder los 255
caracteres' })
  password: string;
}

```

## 3. CreateUserDto (Mejora Final)

typescript

```

import { ApiProperty } from '@nestjs/swagger';
import { IsEmail, IsString, MinLength, MaxLength, Matches, IsNotEmpty
} from 'class-validator';

export class CreateUserDto {
  @ApiProperty({
    description: 'Nombre completo del usuario',
    example: 'Juan Pérez',
    minLength: 3,
    maxLength: 30
  })
  @IsString({ message: 'El nombre debe ser una cadena de texto' })
  @IsNotEmpty({ message: 'El nombre es requerido' })

```

```

    @MinLength(3, { message: 'El nombre debe tener al menos 3
caracteres' })
    @MaxLength(30, { message: 'El nombre no puede exceder los 30
caracteres' })
    name: string;

    @ApiProperty({
        description: 'Email del usuario',
        example: 'usuario@ejemplo.com',
        maxLength: 255
    })
    @IsEmail({}, { message: 'Debe proporcionar un email válido' })
    @NotEmpty({ message: 'El email es requerido' })
    @MaxLength(255, { message: 'El email no puede exceder los 255
caracteres' })
    email: string;

    @ApiProperty({
        description: 'Contraseña del usuario (mínimo 8 caracteres, 1
mayúscula, 1 número)',
        example: 'MiContraseña123',
        minLength: 8,
        maxLength: 255
    })
    @IsString({ message: 'La contraseña debe ser una cadena de texto'
    })
    @NotEmpty({ message: 'La contraseña es requerida' })
    @MinLength(8, { message: 'La contraseña debe tener al menos 8
caracteres' })
    @MaxLength(255, { message: 'La contraseña no puede exceder los 255
caracteres' })
    @Matches(/(?!.*[A-Z])(?!.*[0-9]).*/, {
        message: 'La contraseña debe contener al menos una mayúscula y
un número'
    })
    password: string;
}

```

## Observaciones y mejoras realizadas:

1. **Consistencia en longitudes:**
  - Todos los campos tienen `maxLength` consistentes con la base de datos
  - El nombre tiene máximo 30 caracteres en todos los DTOs
2. **Documentación mejorada:**
  - Formato consistente en `ApiProperty` y `ApiPropertyOptional`
  - Ejemplos claros y descriptivos
  - Especificación de mínimos y máximos en la documentación Swagger
3. **Validaciones robustas:**
  - Validación de formato de email
  - Requisitos complejos de contraseña
  - Mensajes de error descriptivos
4. **Correcciones menores:**
  - Arreglado typo en mensaje de validación ("contraseña")
  - Ajuste de formato para mejor legibilidad

## Recomendación final:

Para hacer estos DTOs aún más robustos, podrías considerar:

### 1. Transformación de datos:

typescript

- `@Transform(({ value }) => value?.trim())`

- **Expresiones regulares mejoradas** para nombres:

typescript

2. `@Matches (/^[a-zA-ZáéíóúÁÉÍÓÚñÑüÿ\s'-]+$/)`

3. **Validación asíncrona** para verificar unicidad de email (usando class-validator custom validator)

¿Te gustaría que desarrolle alguna de estas mejoras adicionales?

**Vale podríamos mejorar los matches para que también fuese obligatorio incluir al menos un símbolo y que hubiese un mensaje personalizado por cada cosa que falte? es decir, al menos una mayúscula, al menos un número al menos un caracter especial dependiendo de qué falta**

## Validación Mejorada de Contraseñas con Mensajes Personalizados

Vamos a mejorar la validación de contraseñas para que:

1. Requiera al menos una mayúscula
2. Requiera al menos un número
3. Requiera al menos un carácter especial
4. Proporcione mensajes de error específicos para cada caso

### Implementación mejorada:

typescript

```
import { ApiProperty, ApiPropertyOptional } from '@nestjs/swagger';
import { IsString, MinLength, MaxLength, IsOptional, Validate,
ValidationArguments, ValidatorConstraint, ValidatorConstraintInterface
} from 'class-validator';
```

```
@ValidatorConstraint({ name: 'passwordRequirements', async: false })
export class PasswordRequirementsConstraint implements
ValidatorConstraintInterface {
  validate(password: string, args: ValidationArguments) {
    // Verificar todos los requisitos
    const hasUpperCase = /[A-Z]/.test(password);
    const hasNumber = /[0-9]/.test(password);
    const hasSpecialChar = /[!@#$%^&*(),.?":{}|<>]/.test(password);

    // Almacenar qué requisitos faltan
    const missingRequirements = [];
```

```

        if (!hasUpperCase) missingRequirements.push('una mayúscula');
        if (!hasNumber) missingRequirements.push('un número');
        if (!hasSpecialChar) missingRequirements.push('un carácter
especial');

        // Si faltan requisitos, establecer el mensaje personalizado
        if (missingRequirements.length > 0) {
            args.constraints[0] = `La contraseña debe contener al menos
${missingRequirements.join(', ')}`;
            return false;
        }

        return true;
    }

    defaultMessage(args: ValidationArguments) {
        return args.constraints[0] || 'La contraseña no cumple con los
requisitos';
    }
}

// Aplicar a los DTOs (ejemplo en CreateUserDto)
export class CreateUserDto {
    // ... otras propiedades

    @ApiProperty({
        description: 'Contraseña (mín 8 caracteres, 1 mayúscula, 1 número,
1 especial)',
        example: 'MiContraseña123!',
        minLength: 8,
        maxLength: 255
    })
    @IsString({ message: 'La contraseña debe ser una cadena de texto' })
    @NotEmpty({ message: 'La contraseña es requerida' })
    @MinLength(8, { message: 'La contraseña debe tener al menos 8
caracteres' })
    @MaxLength(255, { message: 'La contraseña no puede exceder los 255
caracteres' })
    @Validate(PasswordRequirementsConstraint, [''], {
        message: 'Requisitos de contraseña no cumplidos'
    })
    password: string;
}

```

## Cómo funciona:

1. **Validator personalizado:** Creamos PasswordRequirementsConstraint que verifica:
  - Presencia de mayúsculas ([A-Z])
  - Presencia de números ([0-9])
  - Presencia de caracteres especiales ([!@#\$%^&\*() , . ? " : { } | < > ])
2. **Mensajes dinámicos:**
  - Detecta exactamente qué requisitos faltan
  - Construye mensajes como:
    - "La contraseña debe contener al menos una mayúscula, un número"
    - "La contraseña debe contener al menos un carácter especial"

- "La contraseña debe contener al menos una mayúscula, un número, un carácter especial"
3. **Implementación en DTOs:**
- Usamos `@Validate()` en lugar de `@Matches()`
  - Mantenemos todas las otras validaciones (longitud, etc.)

## Ventajas:

1. **Experiencia de usuario mejorada:** Mensajes claros sobre qué falta
2. **Seguridad reforzada:** Requisitos más estrictos para contraseñas
3. **Mantenibilidad:** Lógica de validación centralizada
4. **Reutilizable:** El mismo validador puede usarse en todos los DTOs

## Ejemplo de mensajes de error:

- Si falta mayúscula y número:  
"La contraseña debe contener al menos una mayúscula, un número"
- Si solo falta carácter especial:  
"La contraseña debe contener al menos un carácter especial"

¿Quieres que ajuste algún aspecto específico de esta implementación?

vale no, lo veo demasiado complejo. Hagamos otra cosa. simplemente añade a mi actual `matches @Matches(/(?=.*[A-Z])(?=.*[0-9]).*/, { message: 'La contraseña debe contener al menos una mayúscula y un número' })` que deba tener un caracter especial y fuera, ya desde el front me encargaré de hacer la mostración de mensajes personalizados si hace falta

# Validación Simplificada de Contraseña con Caracteres Especiales

Entiendo que prefieres mantenerlo simple. Aquí está la versión mejorada de tu `@Matches` actual que incluye el requisito de caracteres especiales, manteniendo la simplicidad:

```
typescript
import { ApiProperty } from '@nestjs/swagger';
import { IsString, MinLength, MaxLength, Matches } from 'class-validator';

export class CreateUserDto {
  // ... otras propiedades

  @ApiProperty({
    description: 'Contraseña (mín 8 caracteres, 1 mayúscula, 1 número, 1 especial)',
    example: 'MiContraseña123!',
    minLength: 8,
    maxLength: 255
  })
}
```

```

@IsString()
@MinLength(8)
@MaxLength(255)
@Matches(/(?=[A-Z])(?=[0-9])(?=.*[!@#$%^&*() , . ? " : { } | < >]).*/ , {
  message: 'La contraseña debe contener al menos: 1 mayúscula, 1
número y 1 carácter especial (!@#$%^&* etc.)'
}))
password: string;
}

```

## Características:

1. **Expresión regular mejorada** que verifica:
  - o `(?=[A-Z])`: Al menos una mayúscula
  - o `(?=[0-9])`: Al menos un número
  - o `(?=.*[!@#$%^&*() , . ? " : { } | < >])`: Al menos un carácter especial
2. **Mensaje claro** que enumera todos los requisitos:
  - o "La contraseña debe contener al menos: 1 mayúscula, 1 número y 1 carácter especial (!@#\$%^&\* etc.)"
3. **Mantenimiento simple**:
  - o Una sola regla de validación
  - o Fácil de modificar si cambian los requisitos

## Lista de caracteres especiales permitidos:

! @ # \$ % ^ & \* ( ) , . ? " : { } | < >

Esta solución mantiene la simplicidad que buscas mientras añade el requisito de seguridad adicional de caracteres especiales. El frontend puede manejar la presentación de mensajes más detallados si es necesario.

`[!'!+`@#$%^&*() , . ? " : { } | < >]` además aquí todos los símbolos especiales del teclado

# Validación de Contraseña con Todos los Símbolos Especiales

Aquí tienes la versión completa con todos los caracteres especiales del teclado español/internacional:

```

typescript
import { ApiProperty } from '@nestjs/swagger';
import { IsString, MinLength, MaxLength, Matches } from 'class-validator';

export class CreateUserDto {
  // ... otras propiedades

  @ApiProperty({
    description: 'Contraseña (mín 8 caracteres, 1 mayúscula, 1 número, 1 especial)',
    example: 'P@ssw0rd_2023!',
  })
  password: string;
}

```



```

    minLength: 8,
    maxLength: 255
  })
  @IsString()
  @MinLength(8)
  @MaxLength(255)
  @Matches(/(?=[A-Z])(?=[0-9])(?=.*[!;"#$%&'()*+,-./:;<=>¿?@[\\]\^_`{|}~]).*/) {
    message: 'La contraseña debe contener al menos: 1 mayúscula, 1
    número y 1 carácter especial (!@#$$%* etc.)'
  })
  password: string;
}

```

## Lista completa de caracteres especiales incluidos:

```

text
! ; " # $ % & ' ( ) * + , - . / : ; < = > ¿ ? @ [ \ ] ^ _ ` { | } ~

```

## Explicación de la expresión regular:

1. `(?=.*[A-Z])` - Al menos una letra mayúscula
2. `(?=.*[0-9])` - Al menos un número
3. `(?=.*[!;"#$%&'()*+,-./:;<=>¿?@[\\]\^_`{|}~])`` - Al menos un carácter especial de la lista completa

## Características adicionales:

1. **Incluye símbolos del teclado español** (í, ñ)
2. **Todos los símbolos estándar** de teclados internacionales
3. **Escape adecuado** de caracteres especiales en la regex
4. **Mensaje claro** pero conciso para el usuario

Esta solución mantiene la simplicidad que buscas mientras ofrece la máxima flexibilidad en caracteres especiales permitidos.

Añade una minúscula obligatoria también

# Validación Completa de Contraseña con Todos los Requisitos

Aquí tienes la versión definitiva que incluye todos los requisitos de seguridad para la contraseña:

```

typescript
import { ApiProperty } from '@nestjs/swagger';
import { IsString, MinLength, MaxLength, Matches } from 'class-validator';

export class CreateUserDto {

```

```
// ... otras propiedades

@ApiProperty({
  description: 'Contraseña (mín 8 caracteres, 1 mayúscula, 1
minúscula, 1 número, 1 especial)',
  example: 'P@ssw0rd_2023!',
  minLength: 8,
  maxLength: 255
})
@IsString()
@MinLength(8)
@MaxLength(255)
@Matches(/(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[!;"#$%&'()*+,-
./:;<=>¿?@[\\]^_`{|}~)).*/ , {
  message: 'La contraseña debe contener al menos: 1 mayúscula, 1
minúscula, 1 número y 1 carácter especial (!@#$%&* etc.)'
})
password: string;
}
```

## Requisitos completos de la contraseña:

1. **Longitud:** 8-255 caracteres
2. **Letras minúsculas:** Al menos una ([a-z])
3. **Letras mayúsculas:** Al menos una ([A-Z])
4. **Números:** Al menos uno ([0-9])
5. **Caracteres especiales:** Al menos uno de estos:

text

```
5. ! " # $ % & ' ( ) * + , - . / : ; < = > ¿ ? @ [ \ ] ^ _ ` { | } ~
```

## Mejoras realizadas:

1. **Añadida validación de minúsculas** con (?.\*[a-z])
2. **Actualizado el mensaje de error** para reflejar el nuevo requisito
3. **Mantenida la lista completa** de caracteres especiales
4. **Ejemplo actualizado** para mostrar todos los requisitos

## Expresión regular explicada:

- (?.\*[a-z]) → Debe contener al menos una minúscula
- (?.\*[A-Z]) → Debe contener al menos una mayúscula
- (?.\*[0-9]) → Debe contener al menos un número
- (?.\*[!;"#\$%&'()\*+,-./:;<=>¿?@[\\]^\_`{|}~])` → Debe contener al menos un carácter especial
- .\* → Puede contener cualquier otro carácter (letras, números, símbolos)

Esta implementación ofrece un balance perfecto entre seguridad y simplicidad, cumpliendo con los estándares actuales de seguridad para contraseñas.

hay que cambiar algo del matches en el update por ser opcional o funcionará bien así?

```
import { ApiProperty, ApiPropertyOptional } from '@nestjs/swagger';
import { IsEmail, IsString, MinLength, MaxLength, IsOptional, Matches } from 'class-validator';
export class UpdateUserDto {
  @ApiPropertyOptional({
    description: 'Nuevo nombre del usuario',
    example: 'María15',
    minLength: 3,
    maxLength: 50
  })
  @IsOptional()
  @IsString({ message: 'El nombre debe ser una cadena de texto' })
  @MinLength(3, { message: 'El nombre debe tener al menos 3 caracteres' })
  @MaxLength(50, { message: 'El nombre no puede exceder los 50 caracteres' })
  name?: string;
  @ApiPropertyOptional({
    description: 'Nueva contraseña (mínimo 8 caracteres, 1 mayúscula, 1 número)',
    example: 'NuevaContraseña456',
    minLength: 8,
    maxLength: 255
  })
  @IsOptional()
  @IsString({ message: 'La contraseña debe ser una cadena de texto' })
  @MinLength(8, { message: 'La contraseña debe tener al menos 8 caracteres' })
  @MaxLength(255, { message: 'La contraseña no puede exceder los 255 caracteres' })
  @Matches(/(=?.*[a-z])(=?.*[A-Z])(=?.*[0-9])(=?.*[!;"#$%&'()*+,-./:;<=>¿?@[\\]^_`{|}~]).*/ , { message: 'La contraseña debe contener al menos: 1 minúscula, 1 mayúscula, 1 número y 1 carácter especial'})
  password?: string;
```

## Validación de Contraseña Opcional en UpdateUserDto

Para el UpdateUserDto, donde la contraseña es opcional, la implementación actual es correcta pero podemos mejorarla ligeramente para manejar mejor los casos de campos vacíos:

typescript

```
import { ApiPropertyOptional } from '@nestjs/swagger';
import { IsString, MinLength, MaxLength, IsOptional, Matches } from 'class-validator';

export class UpdateUserDto {
  @ApiPropertyOptional({
    description: 'Nuevo nombre del usuario',
    example: 'María15',
    minLength: 3,
    maxLength: 30 // Corregido para que coincida con la validación (50 en mensaje vs 30 en código)
  })
  @IsOptional()
  @IsString({ message: 'El nombre debe ser una cadena de texto' })
  @MinLength(3, { message: 'El nombre debe tener al menos 3 caracteres' })
  @MaxLength(30, { message: 'El nombre no puede exceder los 30 caracteres' }) // Ajustado a 30
  name?: string;

  @ApiPropertyOptional({
    description: 'Nueva contraseña (mín 8 caracteres, 1 mayúscula, 1 minúscula, 1 número, 1 especial)',
    example: 'Nueva@Contraseña123',
    minLength: 8,
    maxLength: 255
  })
  password?: string;
```

```

    @IsOptional()
    @IsString({ message: 'La contraseña debe ser una cadena de texto'
    })
    @MinLength(8, { message: 'La contraseña debe tener al menos 8
    caracteres' })
    @MaxLength(255, { message: 'La contraseña no puede exceder los 255
    caracteres' })
    @Matches(/(?!.*[a-z])(?!.*[A-Z])(?!.*[0-9])(?!.*[!;"#$%&'()*+,-
    ./:;<=>¿?@[\\]\^\_`{|}~]).*/ , {
        message: 'La contraseña debe contener: 1 minúscula, 1
        mayúscula, 1 número y 1 carácter especial',
        groups: ['update'] // Grupos de validación opcionales
    })
    password?: string;
}

```

## Puntos clave:

- Funcionamiento con @IsOptional():**
  - La validación @Matches solo se aplicará si se proporciona una contraseña
  - Si el campo está vacío o es undefined, no se validará
- Mejoras realizadas:**
  - Corregida inconsistencia entre el MaxLength (30 en validación vs 50 en mensaje)
  - Actualizado el ejemplo de contraseña para que cumpla todos los requisitos
  - Mensaje de descripción más completo
  - Añadido groups para mayor flexibilidad (opcional)
- Comportamiento esperado:**

typescript

```

• // Caso 1: Sin password - válido
const dto1 = new UpdateUserDto();
validate(dto1); // OK

// Caso 2: Con password válido - válido
const dto2 = new UpdateUserDto();
dto2.password = "Nueva@Contraseña123";
validate(dto2); // OK

// Caso 3: Con password inválido - error
const dto3 = new UpdateUserDto();
dto3.password = "debil";
validate(dto3); // ERROR con mensaje descriptivo

```

### • Recomendación adicional (opcional):

Si quieres asegurarte de que cuando se envía el campo no esté vacío (pero siga siendo opcional):

typescript

```

4. @Matches(/(?!.*[a-z])(?!.*[A-Z])(?!.*[0-9])(?!.*[!;"#$%&'()*+,-
   ./:;<=>¿?@[\\]\^\_`{|}~]).*/ , {
5.     message: '...',

```

```

6.      skipEmptyStrings: true // Solo validar si no es string vacío
7. })

```

Esta implementación es correcta y funcionará como esperas para el caso de actualización donde la contraseña es opcional.

```

@Matches(/(?=[a-z])(?=[A-Z])(?=[0-9])(?=.*[!;"#$%&'()*+,-
./:;<=>¿?@[\\]^_`{|}~]).*/, { message: 'La contraseña debe contener al menos: 1
minúscula, 1 mayúscula, 1 número y 1 carácter especial', skipEmptyStrings: true
}) password?: string; @Matches(/(?=[a-z])(?=[A-Z])(?=[0-
9])(?=.*[!;"#$%&'()*+,-./:;<=>¿?@[\\]^_`{|}~]).*/, { message: 'La contraseña debe
contener al menos: 1 minúscula, 1 mayúscula, 1 número y 1 carácter especial',
skipEmptyStrings: true }) password?: string;

```

## Validación Óptima para Contraseña Opcional

La implementación que has compartido es correcta y funciona bien para el caso de contraseña opcional en el `UpdateUserDto`. Aquí está la versión final optimizada:

typescript

```



import { ApiPropertyOptional } from '@nestjs/swagger';
import { IsString, MinLength, MaxLength, IsOptional, Matches } from
'class-validator';

```

```

export class UpdateUserDto {
  @ApiPropertyOptional({
    description: 'Nuevo nombre del usuario (3-30 caracteres)',
    example: 'María15'

```

No overload matches this call. Overload 1 of 2, '(pattern: RegExp, validationOptions?: ValidationOptions): PropertyDecorator', gave the following error. Object literal may only specify known properties, and 'skipEmptyStrings' does not exist in type 'ValidationOptions'. Overload 2 of 2, '(pattern: string, modifiers?: string, validationOptions?: ValidationOptions): PropertyDecorator', gave the following error. Argument of type 'RegExp' is not assignable to parameter of type 'string'.ts(2769)    
Generate docs (Ctrl+.) (alias) Matches(pattern: RegExp, validationOptions?: ValidationOptions): PropertyDecorator (+1 overload) import Matches

## Solución Correcta para Validación de Contraseña Opcional

El error ocurre porque `skipEmptyStrings` no es una opción válida directamente en el decorador `@Matches`. Para manejar correctamente los campos opcionales, debemos usar `@IsOptional()` junto con la validación normal. Aquí está la solución correcta:

typescript

```

import { ApiPropertyOptional } from '@nestjs/swagger';

```

```
import { IsString, MinLength, MaxLength, IsOptional, Matches } from
'class-validator';

export class UpdateUserDto {
  @ApiPropertyOptional({
    description: 'Nueva contraseña (mín 8 caracteres, 1 mayúscula,
1 minúscula, 1 número, 1 especial)',
    example: 'Nueva@Contraseña123',
    minLength: 8,
    maxLength: 255
  })
  @IsOptional()
  @IsString({ message: 'La contraseña debe ser una cadena de texto'
})
  @MinLength(8, { message: 'La contraseña debe tener al menos 8
caracteres' })
  @MaxLength(255, { message: 'La contraseña no puede exceder los 255
caracteres' })
  @Matches(/(?!.*[a-z])(?!.*[A-Z])(?!.*[0-9])(?!.*[!;"#$%&'()*+,-
./:;<=>?@[\]\^\_`{|}~]).*/), {
    message: 'La contraseña debe contener: 1 minúscula, 1
mayúscula, 1 número y 1 carácter especial'
  })
  password?: string;
}
```

## Explicación:

1. **@IsOptional()** es suficiente para hacer que la validación se omita cuando el campo es null o undefined
2. **No se necesita skipEmptyStrings** porque:
  - o @IsOptional() ya maneja este caso
  - o Si necesitas tratar strings vacíos como valores no proporcionados, configura el ValidationPipe globalmente:

typescript

```
// En tu main.ts
app.useGlobalPipes(
  new ValidationPipe({
    skipMissingProperties: true, // Omite validación si propiedad
falta
    whitelist: true,
    forbidNonWhitelisted: true
  })
);
```

3. **La expresión regular** ya está correctamente configurada para validar todos los requisitos de complejidad

## Comportamiento esperado:

- Si password es undefined o null: **No se aplica validación**
- Si password es string vacío "": **Se aplica validación** (fallará por @MinLength(8))
- Si password tiene valor: **Se validan todos los requisitos**

Esta es la forma correcta y más limpia de implementar esta validación en NestJS con class-validator.

Vale a parte de los DTO's lo que faltaría por documentar sería los controladores y ya está, no? No hace falta nada más.

# Documentación Completa de una API NestJS

Efectivamente, además de los DTOs, los controladores son el otro componente clave que debe documentarse. Pero para una documentación API completa y profesional, te recomiendo cubrir estos aspectos:

## 1. Documentación de Controladores (Esencial)

typescript

```
import { Controller, Post, Body } from '@nestjs/common';
import { ApiTags, ApiOperation, ApiResponse } from '@nestjs/swagger';
import { CreateUserDto } from '../dto/create-user.dto';
import { UsersService } from '../users.service';
```

```
@ApiTags('Users') // Agrupa endpoints en Swagger UI
@Controller('users')
export class UsersController {
  constructor(private readonly usersService: UsersService) {}

  @Post()
  @ApiOperation({
    summary: 'Crear nuevo usuario'
  })
```

Hacemos una cosa, te voy pasando mis controladores y tú me lo mandas documentado :)

¡Perfecto! Pásame tus controladores uno por uno y te los devolveré completamente documentados con:

1. Decoradores Swagger estándar
2. Descripciones claras
3. Ejemplos de requests/responses
4. Códigos de estado HTTP relevantes
5. Validaciones adicionales si son necesarias

### Ejemplo de cómo puedes enviármelos:

typescript

```
import { Controller, Get } from '@nestjs/common';
import { UsersService } from '../users.service';

@Controller('users')
export class UsersController {
  constructor(private readonly usersService: UsersService) {}

  @Get()
  findAll() {
```

```

        return this.usersService.findAll();
    }
}

```

## Y te devolveré algo como esto:

### typescript

```

import { Controller, Get } from '@nestjs/common';
import { ApiTags, ApiOperation, ApiResponse, ApiQuery } from
'@nestjs/swagger';
import { UsersService } from '../users.service';
import { User } from '../entities/user.entity';

@ApiTags('Users - Gestion de usuarios')
@Controller('users')
export class UsersController {
    constructor(private readonly usersService: UsersService) {}

    @Get()
    @ApiOperation({
        summary: 'Obtener todos los usuarios',
        description: 'Retorna una lista paginada de todos los usuarios
registrados en el sistema'
    })
    @ApiQuery({
        name: 'page',
        required: false,
        type: Number,
        description: 'Número de página para paginación',
        example: 1
    })
    @ApiResponse({
        status: 200,
        description: 'Lista de usuarios obtenida exitosamente',
        type: [User]
    })
    @ApiResponse({
        status: 500,
        description: 'Error interno del servidor'
    })
    findAll() {
        return this.usersService.findAll();
    }
}

```

**Cuando quieras, pásame tu primer controlador y comenzamos a documentarlo!**  
Puedes enviarme:

1. El código del controlador
2. Algún detalle especial que quieras destacar
3. La estructura de las respuestas (si tienes DTOs de respuesta)
4. Cualquier requisito específico de documentación que necesites

¿Qué controlador quieres que documentemos primero? ☐

```

import { Controller, Get, Post, Body, Param, Patch, UseGuards, Req, Delete }
from '@nestjs/common'; import { UsersService } from '../users.service'; import {

```



```

CreateUserDto } from './dto/create-users.dto'; import { UpdateUserDto } from
'./dto/update-users.dto'; import { JwtModule } from '@nestjs/jwt'; import {
JwtAuthGuard } from '@nestjs/passport'; import { ApiBearerAuth }
from '@nestjs/swagger'; import { LoginUserDto } from './dto/login-user.dto';
@Controller('users') export class UsersController { constructor(private readonly
userService: UsersService) {} @Get() getAll() { return this.userService.getAll(); }
@Get('id/:id') getById(@Param('id') id: number) { return
this.userService.getById(id); } @Get('email/:email')
getEmail(@Param('email') email: string) { return
this.userService.getEmail(email); } @Post() create(@Body() createUserDto:
CreateUserDto) { return this.userService.create(createUserDto); } @Patch('me')
@ApiBearerAuth('JWT-auth') @UseGuards(JwtAuthGuard) update( @Req()
req, @Body() updateUserDto: UpdateUserDto, ) { return
this.userService.update(req.user.sub,updateUserDto); } @Delete('me')
@ApiBearerAuth('JWT-auth') @UseGuards(JwtAuthGuard) async delete(
@Req() req, @Body() loginUserDto: LoginUserDto, ) { await
this.userService.delete(req.user.email,loginUserDto); return { success: true,
message: `User deleted successfully`, }; } }

```

## Controlador de Usuarios Documentado

typescript

```

import { Controller, Get, Post, Body, Param, Patch, UseGuards, Req,
Delete } from '@nestjs/common';
import {
  ApiTags,
  ApiOperation,
  ApiResponse,
  ApiParam,
  ApiBody,
  ApiBearerAuth,
  ApiConflictResponse,
  ApiNotFoundResponse,
  ApiUnauthorizedResponse
} from '@nestjs/swagger';
import { UsersService } from '../users.service';
import { CreateUserDto } from '../dto/create-users.dto';
import { UpdateUserDto } from '../dto/update-users.dto';
import { JwtAuthGuard } from '@nestjs/passport';
import { LoginUserDto } from '../dto/login-user.dto';
import { User } from '../entities/user.entity';

@ApiTags('Users - Gestión de usuarios')
@Controller('users')
export class UsersController {
  constructor(private readonly userService: UsersService) {}

  @Get()
  @ApiOperation({
    summary: 'Obtener todos los usuarios',
    description: 'Retorna una lista completa de todos los usuarios
registrados en el sistema'
  })
  @ApiResponse({
    status: 200,

```

```

        description: 'Lista de usuarios obtenida exitosamente',
        type: [User]
    })
    @ApiResponse({
        status: 500,
        description: 'Error interno del servidor'
    })
    getAll() {
        return this.usersService.getAll();
    }

    @Get('id/:id')
    @ApiOperation({
        summary: 'Obtener usuario por ID',
        description: 'Retorna un usuario específico basado en su ID
numérico'
    })
    @ApiParam({
        name: 'id',
        type: Number,
        description: 'ID del usuario',
        example: 1
    })
    @ApiResponse({
        status: 200,
        description: 'Usuario encontrado',
        type: User
    })
    @ApiNotFoundResponse({
        description: 'Usuario no encontrado'
    })
    getById(@Param('id') id: number) {
        return this.usersService.getById(id);
    }

    @Get('email/:email')
    @ApiOperation({
        summary: 'Obtener usuario por email',
        description: 'Retorna un usuario específico basado en su
dirección de email'
    })
    @ApiParam({
        name: 'email',
        type: String,
        description: 'Email del usuario',
        example: 'usuario@ejemplo.com'
    })
    @ApiResponse({
        status: 200,
        description: 'Usuario encontrado',
        type: User
    })
    @ApiNotFoundResponse({
        description: 'Usuario no encontrado'
    })
    getByEmail(@Param('email') email: string) {
        return this.usersService.getByEmail(email);
    }

    @Post()
    @ApiOperation({

```

```

        summary: 'Crear nuevo usuario',
        description: 'Registra un nuevo usuario en el sistema'
    ))
    @ApiBody({
        type: CreateUserDto,
        description: 'Datos requeridos para crear un nuevo usuario'
    })
    @ApiResponse({
        status: 201,
        description: 'Usuario creado exitosamente',
        type: User
    })
    @ApiConflictResponse({
        description: 'El email ya está registrado'
    })
    @ApiResponse({
        status: 400,
        description: 'Datos de entrada inválidos'
    })
    create(@Body() createUserDto: CreateUserDto) {
        return this.usersService.create(createUserDto);
    }

    @Patch('me')
    @ApiBearerAuth('JWT-auth')
    @UseGuards(JwtAuthGuard)
    @ApiOperation({
        summary: 'Actualizar usuario actual',
        description: 'Actualiza la información del usuario
autenticado'
    })
    @ApiBody({
        type: UpdateUserDto,
        description: 'Datos a actualizar del usuario'
    })
    @ApiResponse({
        status: 200,
        description: 'Usuario actualizado exitosamente',
        type: User
    })
    @ApiUnauthorizedResponse({
        description: 'No autorizado - Se requiere autenticación'
    })
    @ApiResponse({
        status: 400,
        description: 'Datos de entrada inválidos'
    })
    update(
        @Req() req,
        @Body() updateUserDto: UpdateUserDto,
    ) {
        return this.usersService.update(req.user.sub, updateUserDto);
    }

    @Delete('me')
    @ApiBearerAuth('JWT-auth')
    @UseGuards(JwtAuthGuard)
    @ApiOperation({
        summary: 'Eliminar cuenta de usuario actual',
        description: 'Elimina permanentemente la cuenta del usuario
autenticado. Requiere confirmación con contraseña.'
    })

```

```

    })
    @ApiBody({
      type: LoginUserDto,
      description: 'Credenciales para confirmar la eliminación'
    })
    @ApiResponse({
      status: 200,
      description: 'Usuario eliminado exitosamente',
      schema: {
        example: {
          success: true,
          message: 'User deleted successfully'
        }
      }
    })
    @ApiUnauthorizedResponse({
      description: 'No autorizado - Credenciales inválidas'
    })
    @ApiResponse({
      status: 404,
      description: 'Usuario no encontrado'
    })
    async delete(
      @Req() req,
      @Body() loginUserDto: LoginUserDto,
    ) {
      await this.usersService.delete(req.user.email, loginUserDto);
      return {
        success: true,
        message: `User deleted successfully`,
      };
    }
  }
}

```

## Mejoras realizadas:

1. **Documentación Swagger completa** para cada endpoint
2. **Decoradores específicos** para cada tipo de operación:
  - o `@ApiOperation` para descripciones
  - o `@ApiParam` para parámetros de ruta
  - o `@ApiBody` para estructuras de payload
  - o `@ApiResponse` para respuestas exitosas
  - o `@ApiErrorResponse` para errores comunes
3. **Especificación de tipos** en las respuestas
4. **Ejemplos claros** de parámetros y respuestas
5. **Documentación de seguridad** con `@ApiBearerAuth`
6. **Códigos de estado HTTP** bien definidos

## Puntos destacables:

- La ruta `PATCH /me` y `DELETE /me` quedaron perfectamente documentadas con sus requisitos de autenticación
- Se agregaron respuestas para todos los casos comunes (éxito, errores, validaciones)
- Las descripciones son claras y útiles para los consumidores de la API

- Se mantuvo la estructura limpia del controlador original

¿Te gustaría que hagamos algún ajuste específico a la documentación? ¿O prefieres que documentemos otro controlador?

```
import { Controller, Get, Post, Body, Param, Patch, UseGuards, Req, Delete }
from '@nestjs/common'; import { UsersService } from './users.service'; import {
CreateUserDto } from './dto/create-users.dto'; import { UpdateUserDto } from
'./dto/update-users.dto'; import { JwtModule } from '@nestjs/jwt'; import {
JwtAuthGuard } from '@auth/guards/jwt-auth.guard'; import { ApiBearerAuth,
ApiBody, ApiConflictResponse, ApiNotFoundResponse, ApiOperation, ApiParam,
ApiResponse, ApiTags, ApiUnauthorizedResponse } from '@nestjs/swagger';
import { LoginUserDto } from './dto/login-user.dto'; import { Users } from
'./entities/users.entity'; @ApiTags('Users - Gestión de usuarios')
@Controller('users') export class UsersController { constructor(private readonly
userService: UsersService) {} @Get() @ApiOperation({ summary: 'Obtener todos
los usuarios', description: 'Retorna una lista completa de todos los usuarios
registrados en el sistema' }) @ApiResponse({ status: 200, description: 'Lista de
usuarios obtenida exitosamente', type: [Users] }) @ApiResponse({ status: 500,
description: 'Error interno del servidor' }) getAll() { return
this.userService.getAll(); } @Get('id/:id') @ApiOperation({ summary: 'Obtener
usuario por ID', description: 'Retorna un usuario específico basado en su ID
numérico' }) @ApiParam({ name: 'id', type: Number, description: 'ID del
usuario', example: 1 }) @ApiResponse({ status: 200, description: 'Usuario
encontrado', type: Users }) @ApiNotFoundResponse({ description: 'Usuario no
encontrado' }) getId(@Param('id') id: number) { return
this.userService.getId(id); } @Get('email/:email') @ApiOperation({ summary:
'Obtener usuario por email', description: 'Retorna un usuario específico basado en
su dirección de email' }) @ApiParam({ name: 'email', type: String, description:
'Email del usuario', example: 'usuario@ejemplo.com' }) @ApiResponse({ status:
200, description: 'Usuario encontrado', type: Users }) @ApiNotFoundResponse({
description: 'Usuario no encontrado' }) getEmail(@Param('email') email:
string) { return this.userService.getEmail(email); } @Post() @ApiOperation({
summary: 'Crear nuevo usuario', description: 'Registra un nuevo usuario en el
sistema' }) @ApiBody({ type: CreateUserDto, description: 'Datos requeridos para
crear un nuevo usuario' }) @ApiResponse({ status: 201, description: 'Usuario
creado exitosamente', type: Users }) @ApiConflictResponse({ description: 'El
email ya está registrado' }) @ApiResponse({ status: 400, description: 'Datos de
entrada inválidos' }) create(@Body() createUserDto: CreateUserDto) { return
this.userService.create(createUserDto); } @Patch('me') @ApiBearerAuth('JWT-
auth') @UseGuards(JwtAuthGuard) @ApiOperation({ summary: 'Actualizar
usuario actual', description: 'Actualiza la información del usuario autenticado' })
@ApiBody({ type: UpdateUserDto, description: 'Datos a actualizar del usuario' })
@ApiResponse({ status: 200, description: 'Usuario actualizado exitosamente',
type: Users }) @ApiUnauthorizedResponse({ description: 'No autorizado - Se
requiere autenticación' }) @ApiResponse({ status: 400, description: 'Datos de
entrada inválidos' }) update( @Req() req, @Body() updateUserDto:
UpdateUserDto, ) { return this.userService.update(req.user.sub, updateUserDto); }
@Delete('me') @ApiBearerAuth('JWT-auth') @UseGuards(JwtAuthGuard)
@ApiOperation({ summary: 'Eliminar cuenta de usuario actual', description:
```

'Elimina permanentemente la cuenta del usuario autenticado. Requiere confirmación con contraseña.')} @ApiBody({ type: LoginUserDto, description: 'Credenciales para confirmar la eliminación' }) @ApiResponse({ status: 200, description: 'Usuario eliminado exitosamente', schema: { example: { success: true, message: 'User deleted successfully' } } }) @ApiUnauthorizedResponse({ description: 'No autorizado - Credenciales inválidas' }) @ApiResponse({ status: 404, description: 'Usuario no encontrado' }) async delete( @Req() req, @Body() loginUserDto: LoginUserDto, ) { await this.usersService.delete(req.user.email,loginUserDto); return { success: true, message: `User deleted successfully`, }; } } Observa la manera en que he dejado los espacios y he ido saltando líneas. Me gustaría que los replicaras en los siguientes. Además, no tienes por qué enviarme todo el código solo los bloques de documentación, te parece bien?

## Bloques de Documentación con Formato Estilo

Entiendo perfectamente el formato que prefieres. Aquí tienes los bloques de documentación con el espaciado y estructura que solicitas:

typescript

```
@Get()
@ApiOperation({
  summary: 'Obtener todos los usuarios',
  description: 'Retorna una lista completa de todos los usuarios
registrados en el sistema'
})
@ApiResponse({
  status: 200,
  description: 'Lista de usuarios obtenida exitosamente',
  type: [Users]
})
@ApiResponse({
  status: 500,
  description: 'Error interno del servidor'
})
```

typescript

```
@Get('id/:id')
@ApiOperation({
  summary: 'Obtener usuario por ID',
  description: 'Retorna un usuario específico basado en su ID
numérico'
})
@ApiParam({
  name: 'id',
  type: Number,
  description: 'ID del usuario',
  example: 1
})
@ApiResponse({
  status: 200,
  description: 'Usuario encontrado',
  type: Users
})
```

```
@ApiNotFoundResponse({
  description: 'Usuario no encontrado'
})
```

**typescript**

```
@Get('email/:email')
@ApiOperation({
  summary: 'Obtener usuario por email',
  description: 'Retorna
```

**NO. Quizás no se aprecia, pero aquí por ejemplo @ApiBody({ type: UpdateUserDto, description: 'Datos a actualizar del usuario' }) @ApiResponse({ status: 200, description: 'Usuario actualizado exitosamente', type: Users }) @ApiUnauthorizedResponse({ description: 'No autorizado - Se requiere autenticación' }) @ApiResponse({ status: 400, description: 'Datos de entrada inválidos' })** Va todo lo del mismo **@** en la misma línea, salvo que haya demasiados caracteres entonces sí salto abajo. Además, las llaves y paréntesis de cierre no ocupan una nueva línea.

Entiendo perfectamente el formato que necesitas. Aquí tienes los bloques de documentación con el estilo exacto que buscas:

**typescript**

```
@Get()
@ApiOperation({ summary: 'Obtener todos los usuarios', description:
'Returna una lista completa de todos los usuarios registrados en el
sistema'})
@ApiResponse({ status: 200, description: 'Lista de usuarios obtenida
exitosamente', type: [Users]})
@ApiResponse({ status: 500, description: 'Error interno del
servidor'})
```

**typescript**

```
@Get('id/:id')
@ApiOperation({ summary: 'Obtener usuario por ID', description:
'Returna un usuario específico basado en su ID numérico'})
@ApiParam({ name: 'id', type: Number, description
Veo que lo has entendido perfectamente. Genial, lo dicho, vamos al siguiente controlador. Hazlo así por favor. import { Controller, Get, Post, Body, Patch, Param, Delete, Req, UseGuards } from '@nestjs/common'; import { CategoryService } from './category.service'; import { CreateUpdateCategoryDto } from './dto/createUpdate-category.dto'; import { ApiBearerAuth } from '@nestjs/swagger'; import { JwtAuthGuard } from '@auth/guards/jwt-auth.guard'; @Controller('category') export class CategoryController { constructor(private readonly categoryService: CategoryService) {} @ApiBearerAuth('JWT-auth') @UseGuards(JwtAuthGuard) @Post() create( @Req() req, @Body() createUpdateCategoryDto: CreateUpdateCategoryDto, ) { return this.categoryService.create(req.user.sub, createUpdateCategoryDto); } @ApiBearerAuth('JWT-auth') @UseGuards(JwtAuthGuard) @Get('user/') getByUser( @Req() req ) { return this.categoryService.getByUser(req.user.sub); } @ApiBearerAuth('JWT-auth') @UseGuards(JwtAuthGuard) @Get('/:id') getById( @Req() req, @Param('id') id: number ) { return this.categoryService.getById(req.user.sub, id); } @ApiBearerAuth('JWT-auth') @UseGuards(JwtAuthGuard) @Patch('/:id') async update( @Req() req, @Param('id') id: number, @Body() createUpdateCategoryDto: CreateUpdateCategoryDto ) { return await this.categoryService.update(req.user.sub, id, createUpdateCategoryDto); }
```

```

@ApiBearerAuth('JWT-auth') @UseGuards(JwtAuthGuard) @Delete(':id') async
delete( @Req() req, @Param('id') id: number, ) { await
this.categoryService.delete(req.user.sub, id); return { success: true, message:
`Category ${id} deleted successfully`, deletedId: id }; } }

```

## Controlador de Categorías Documentado

typescript

```

import { Controller, Get, Post, Body, Patch, Param, Delete, Req,
UseGuards } from '@nestjs/common';
import { CategoryService } from '../category.service';
import { CreateUpdateCategoryDto } from '../dto/createUpdate-
category.dto';
import { ApiBearerAuth, ApiOperation, ApiResponse, ApiParam, ApiBody,
ApiNotFoundResponse, ApiUnauthorizedResponse, ApiTags } from
'@nestjs/swagger';
import { JwtAuthGuard } from '@auth/guards/jwt-auth.guard';
import { Category } from '../entities/category.entity';

@ApiTags('Categories - Gestión de categorías')
@Controller('category')
export class CategoryController {
  constructor(private readonly categoryService: CategoryService) {}

  @ApiBearerAuth('JWT-auth')
  @UseGuards(JwtAuthGuard)
  @Post()
  @ApiOperation({ summary: 'Crear nueva categoría', description: 'Crea
una nueva categoría asociada al usuario autenticado' })
  @ApiBody({ type: CreateUpdateCategoryDto, description: 'Datos
requeridos para crear la categoría' })
  @ApiResponse({ status: 201, description: 'Categoría creada
exitosamente', type: Category })
  @ApiUnauthorizedResponse({ description: 'No autorizado - Se requiere
autenticación' })
  @ApiResponse({ status: 400, description: 'Datos de entrada
inválidos' })
  create(
    @Req() req,
    @Body() createUpdateCategoryDto: CreateUpdateCategoryDto,
  ) {
    return this.categoryService.create(req.user.sub,
createUpdateCategoryDto);
  }

  @ApiBearerAuth('JWT-auth')
  @UseGuards(JwtAuthGuard)
  @Get('user/')
  @ApiOperation({ summary: 'Obtener categorías del usuario',
description: 'Retorna todas las categorías del usuario autenticado' })
  @ApiResponse({ status: 200, description: 'Lista de categorías
obtenida exitosamente', type: [Category] })
  @ApiUnauthorizedResponse({ description: 'No autorizado - Se requiere
autenticación' })
  getByUser(@Req() req) {
    return this.categoryService.getByUser(req.user.sub);
  }
}

```



```

    @ApiBearerAuth('JWT-auth')
    @UseGuards(JwtAuthGuard)
    @Get('/:id')
    @ApiOperation({ summary: 'Obtener categoría por ID', description:
'Returna una categoría específica del usuario autenticado'})
    @ApiParam({ name: 'id', type: Number, description: 'ID de la
categoría', example: 1})
    @ApiResponse({ status: 200, description: 'Categoría encontrada',
type: Category})
    @ApiNotFoundResponse({ description: 'Categoría no encontrada'})
    @ApiUnauthorizedResponse({ description: 'No autorizado - Se requiere
autenticación'})
    getById(
        @Req() req,
        @Param('id') id: number
    ) {
        return this.categoryService.getById(req.user.sub, id);
    }

    @ApiBearerAuth('JWT-auth')
    @UseGuards(JwtAuthGuard)
    @Patch('/:id')
    @ApiOperation({ summary: 'Actualizar categoría', description:
'Actualiza una categoría específica del usuario autenticado'})
    @ApiParam({ name: 'id', type: Number, description: 'ID de la
categoría a actualizar', example: 1})
    @ApiBody({ type: CreateUpdateCategoryDto, description: 'Datos a
actualizar de la categoría'})
    @ApiResponse({ status: 200, description: 'Categoría actualizada
exitosamente', type: Category})
    @ApiNotFoundResponse({ description: 'Categoría no encontrada'})
    @ApiUnauthorizedResponse({ description: 'No autorizado - Se requiere
autenticación'})
    @ApiResponse({ status: 400, description: 'Datos de entrada
inválidos'})
    async update(
        @Req() req,
        @Param('id') id: number,
        @Body() createUpdateCategoryDto: CreateUpdateCategoryDto) {
        return await this.categoryService.update(req.user.sub, id,
createUpdateCategoryDto);
    }

    @ApiBearerAuth('JWT-auth')
    @UseGuards(JwtAuthGuard)
    @Delete('/:id')
    @ApiOperation({ summary: 'Eliminar categoría', description: 'Elimina
una categoría específica del usuario autenticado'})
    @ApiParam({ name: 'id', type: Number, description: 'ID de la
categoría a eliminar', example: 1})
    @ApiResponse({ status: 200, description: 'Categoría eliminada
exitosamente', schema: {
        example: {
            success: true,
            message: 'Category {id} deleted successfully',
            deletedId: 1
        }
    })
    @ApiNotFoundResponse({ description: 'Categoría no encontrada'})
    @ApiUnauthorizedResponse({ description: 'No autorizado - Se requiere
autenticación'})

```

```
async delete(
  @Req() req,
  @Param('id') id: number, ) {
  await this.categoryService.delete(req.user.sub, id);
  return {
    success: true,
    message: `Category ${id} deleted successfully`,
    deletedId: id
  };
}
```