Proyecto final Typescript

Magnolia Barajas

Juan Andres Rico Teran

2025

Politécnico internacional de occidente

**Taller Final Typescrit**

El primer paso consiste en elegir una idea para tu proyecto. Si decides trabajar con productos o servicios, algunos ejemplos comunes pueden ser:

* Proyecto de productos: Una API para gestionar productos en una tienda en línea (con atributos como nombre, precio, descripción, etc.).
* Proyecto de servicios: Una API para gestionar servicios que una empresa ofrece (por ejemplo, reparación de dispositivos, asesoramiento financiero, etc.).

Supongamos que elijes el proyecto de productos, por lo que tu API será para gestionar productos en una tienda.

**2. Crear el Proyecto con NestJS**

**Para crear el proyecto con NestJS, NodeJS, y TypeScript, sigue estos pasos:**

**Paso 1: Instalar el CLI de NestJS**

**Si no tienes instalado NestJS, instala el NestJS CLI:**

bash

CopiarEditar

npm i -g @nestjs/cli

**Paso 2: Crear un Proyecto Nuevo**

**Crea un nuevo proyecto con NestJS usando el comando CLI:**

bash

CopiarEditar

nest new productos-api

Este comando genera una estructura básica de directorios y archivos para tu proyecto.

Paso 3: Instalar Dependencias

Asegúrate de tener todas las dependencias necesarias, incluyendo MongoDB, Validator, y Transformer:

bash

CopiarEditar

npm install @nestjs/mongoose mongoose class-validator class-transformer

**3. Crear el Módulo de Productos**

**Crea el módulo productos dentro de tu proyecto. En NestJS, los módulos son unidades de funcionalidad. Para crear el módulo de productos, ejecuta el siguiente comando en tu terminal:**

bash

CopiarEditar

nest generate module productos

Este comando crea un módulo llamado productos en el directorio src/productos.

**4. Estructura del Módulo: CRUD (Crear, Leer, Actualizar, Eliminar)**

**Dentro del módulo productos, necesitarás crear el controlador, servicio y el esquema para MongoDB.**

**Paso 1: Crear el Controlador de Productos**

**El controlador maneja las solicitudes HTTP para las distintas rutas de la API. Crea el controlador con:**

**bash**

CopiarEditar

nest generate controller productos

Esto genera un archivo productos.controller.ts. En este archivo, agregarás las rutas para listar, buscar, crear, editar y eliminar productos.

Ejemplo de ProductosController:

typescript

CopiarEditar

import { Controller, Get, Post, Body, Param, Patch, Delete } from '@nestjs/common';

import { ProductosService } from './productos.service';

import { CreateProductoDto } from './dto/create-producto.dto';

import { UpdateProductoDto } from './dto/update-producto.dto';

@Controller('productos')

export class ProductosController {

constructor(private readonly productosService: ProductosService) {}

@Get()

findAll() {

return this.productosService.findAll();

}

@Get(':id')

findOne(@Param('id') id: string) {

return this.productosService.findOne(id);

}

@Post()

create(@Body() createProductoDto: CreateProductoDto) {

return this.productosService.create(createProductoDto);

}

@Patch(':id')

update(@Param('id') id: string, @Body() updateProductoDto: UpdateProductoDto) {

return this.productosService.update(id, updateProductoDto);

}

@Delete(':id')

remove(@Param('id') id: string) {

return this.productosService.remove(id);

}

}

**Paso 2: Crear el Servicio de Productos**

**El servicio maneja la lógica del negocio. Para crearlo, usa:**

**bash**

**CopiarEditar**

**nest generate service productos**

**En el archivo productos.service.ts, tendrás las funciones para interactuar con la base de datos y realizar el CRUD de productos.**

**Ejemplo de ProductosService:**

typescript

CopiarEditar

import { Injectable } from '@nestjs/common';

import { InjectModel } from '@nestjs/mongoose';

import { Model } from 'mongoose';

import { Producto, ProductoDocument } from './schemas/producto.schema';

import { CreateProductoDto } from './dto/create-producto.dto';

import { UpdateProductoDto } from './dto/update-producto.dto';

@Injectable()

export class ProductosService {

constructor(@InjectModel(Producto.name) private productoModel: Model<ProductoDocument>) {}

async findAll(): Promise<Producto[]> {

return this.productoModel.find().exec();

}

async findOne(id: string): Promise<Producto | null> {

return this.productoModel.findById(id).exec();

}

async create(createProductoDto: CreateProductoDto): Promise<Producto> {

const createdProducto = new this.productoModel(createProductoDto);

return createdProducto.save();

}

async update(id: string, updateProductoDto: UpdateProductoDto): Promise<Producto | null> {

return this.productoModel.findByIdAndUpdate(id, updateProductoDto, { new: true }).exec();

}

async remove(id: string): Promise<Producto | null> {

return this.productoModel.findByIdAndRemove(id).exec();

}

}

**Paso 3: Crear el Esquema de MongoDB**

Crea un archivo para el esquema de MongoDB en schemas/producto.schema.ts:

typescript

CopiarEditar

import { Schema, Document } from 'mongoose';

export interface Producto extends Document {

name: string;

description: string;

price: number;

quantity: number;

}

export const ProductoSchema = new Schema({

name: { type: String, required: true },

description: { type: String, required: true },

price: { type: Number, required: true },

quantity: { type: Number, required: true },

});

En este caso, se define el esquema para los productos, con campos como name, description, price, y quantity.

**5. Validar Datos con Validator y Transformer**

Utiliza class-validator y class-transformer para validar los datos de entrada en tus DTOs.

Ejemplo de CreateProductoDto con validación:

typescript

CopiarEditar

import { IsString, IsNumber, IsNotEmpty } from 'class-validator';

export class CreateProductoDto {

@IsString()

@IsNotEmpty()

name: string;

@IsString()

@IsNotEmpty()

description: string;

@IsNumber()

@IsNotEmpty()

price: number;

@IsNumber()

@IsNotEmpty()

quantity: number;

}

Ejemplo de UpdateProductoDto:

typescript

CopiarEditar

import { PartialType } from '@nestjs/mapped-types';

import { CreateProductoDto } from './create-producto.dto';

export class UpdateProductoDto extends PartialType(CreateProductoDto) {}

El uso de PartialType hace que todos los campos del DTO sean opcionales, ya que no es necesario actualizar todos los campos de un producto.

**6. Probar la API con Thunder Client**

Con la API lista, puedes usar Thunder Client para probar las rutas que has creado.

1. Abre Thunder Client en Visual Studio Code.
2. Crea una nueva solicitud HTTP para cada operación:
   * GET: Para listar los productos o buscar un producto por ID.
   * POST: Para crear un nuevo producto.
   * PATCH: Para editar un producto existente.
   * DELETE: Para eliminar un producto.

Ejemplo de POST para crear un producto:

* Método: POST
* URL: http://localhost:3000/productos
* Cuerpo de la solicitud (JSON):

json

CopiarEditar

{

"name": "Producto 1",

"description": "Descripción del producto",

"price": 100,

"quantity": 50

}

}

Evidencia

