Prueba de Desempeño – Módulo: Node.js con NestJS y TypeORM

Caso de uso: Sistema de Soporte Técnico – "TechHelpDesk"

Contexto

La empresa **TechHelpDesk** ofrece servicios de soporte técnico para organizaciones que manejan múltiples equipos de trabajo.

Actualmente, las solicitudes de soporte (tickets) se registran manualmente a través de hojas de cálculo, generando retrasos en la atención, pérdida de trazabilidad y duplicidad de reportes.

Objetivo

Bajo tu rol como **desarrollador backend**, deberás construir una **API REST** utilizando **NestJS**, **TypeORM**, **JWT**, y **Swagger**, que permita administrar todo el ciclo de vida de los **tickets de soporte técnico**.

El sistema debe permitir:

- Registrar usuarios con diferentes **roles** (Administrador, Técnico, Cliente).
- Crear, asignar y actualizar **tickets de soporte**.
- Controlar el **estado del ticket** (abierto, en progreso, resuelto, cerrado).
- Controlar las categorías de las incidencias (Solicitud, Incidente de Hardware, Incidente de Software)
- Consultar el **historial de tickets por cliente** y por técnico.

La base de datos debe estar poblada con **seeders** iniciales (usuarios, técnicos, clientes, categorías de incidencias).

Requisitos técnicos

1. Sistema de autenticación y roles

- Implementar autenticación mediante **JWT**.
- Implementar **Guards personalizados** para controlar acceso según el rol.
- Roles:
    - **Administrador:** CRUD completo de usuarios, técnicos, clientes, categorías y tickets.
    - **Técnico:** puede consultar y actualizar el estado de los tickets asignados.
    - **Cliente:** puede registrar nuevos tickets y consultar su historial.
- Utilizar **Decorators personalizados** (`@Roles()`, `@CurrentUser()`) para simplificar la gestión de permisos.

2. Persistencia de datos

- Base de datos relacional **PostgreSQL**.
- Uso de **TypeORM** para manejar entidades, relaciones.
- Crear las siguientes entidades con sus relaciones:
    - **User** (id, name, email, password, role)
    - **Category** (id, name, description)
    - **Ticket** (id, title, description, status, priority, createdAt, updatedAt)
    - **Client** (id, name, company, contactEmail)
    - **Technician** (id, name, specialty, availability)
- La relación entre tickets, técnicos y clientes debe estar correctamente modelada incluyendo Constrains.

3. Validaciones

- Uso de **Pipes** para validar los DTOs de entrada.
- No se debe permitir crear tickets sin categoría ni cliente válido.
- Validar que un técnico no pueda tener más de 5 tickets “en progreso” al mismo tiempo.
- El estado del ticket solo puede ser modificado siguiendo la secuencia: `Abierto → En progreso → Resuelto → Cerrado`.

4. Interceptores

- Implementar un **TransformInterceptor** para formatear las respuestas en un mismo estándar `{ success, data, message }`.

5. Documentación

- Documentar la API con **Swagger** usando el módulo `@nestjs/swagger`.
- Incluir ejemplos de request y response en cada endpoint.

6. CLI de Nest

- Utiliza los comandos del **CLI de Nest** para generar módulos, controladores, servicios y entidades (`nest g module`, `nest g controller`, etc.).
- La estructura del proyecto debe reflejar buenas prácticas de modularización (por dominios: `tickets`, `users`, `auth`, `categories`, etc.).

Criterios de aceptación

Funcionalidad completa

- Los usuarios pueden registrarse, iniciar sesión y operar según su rol.
- El **Administrador** gestiona usuarios, técnicos, clientes, categorías y tickets.
- El **Técnico** consulta y cambia el estado de los tickets asignados.
- El **Cliente** crea nuevos tickets y consulta su historial, podrá consultar todos los tickets o buscar un ticket por un identificador, por lo cual, será necesario en las búsqueda utilizar decoradores de tipo `@param @Get(':id')`

Gestión de tickets

- Endpoint protegido para crear tickets (`POST /tickets`), con validación de cliente y categoría.
- Endpoint para cambiar el estado (`PATCH /tickets/:id/status`) con guard de rol y validaciones lógicas.
- Endpoint para consultar historial de tickets por cliente (`GET /tickets/client/:id`).
- Endpoint para listar tickets por técnico (`GET /tickets/technician/:id`).

Gestión de usuarios y categorías

- Endpoints protegidos (`/users`, `/categories`) para CRUD completos, accesibles solo a Administradores.

Validaciones y Pipes

- Los DTOs usan `class-validator` para campos obligatorios.
- Las excepciones deben manejarse con `HttpException` y `ExceptionFilter` personalizados.

Clean Code

- El código debe usar inyección de dependencias, tipado con TypeScript y estructura modular.
- Se deben aplicar los principios SOLID en todas la lógica implementada.

Pruebas Unitarias (Jest) - Obligatorio

- Implementar al menos dos pruebas unitarias con **Jest**:
  - Una para la creación de tickets.
  - Otra para el cambio de estado.
- Cobertura mínima del 40% (`npm run test:cov`).

Entregables

1. Enlace al **repositorio público en GitHub**.
2. Archivo **README.md** con:
   - Nombre del coder y clan.
   - Instrucciones para levantar el proyecto
   - URL del Swagger y ejemplos de endpoints.
3. Archivo `.sql` o `dump` de la base de datos con datos iniciales.
4. Ejemplo del archivo `.env` en el README.

Puntos extras (Opcionales)

Despliegue con Docker

- Crear un **Dockerfile** para construir la imagen de la API.
- Crear un **docker-compose.yml** que levante:
    - Contenedor de la API NestJS.
    - Contenedor de PostgreSQL con volumen persistente.

Recursos sugeridos

- Documentación oficial de NestJS
- Documentación de TypeORM
- Documentación de Swagger para Nest
- Documentación de Jest
- Documentación de Docker