Pregunta:

Muchas gracias por toda esta información, estoy aprendiendo muchísimo, aunque no le estoy reteniendo todo, son muchos conceptos.

Ya que tengo esta estructura montada quiero comenzar a utilizar una nueva base de datos basada en MariaDB y que ya estára ubicada en mi localhost (no en la ip de la aplicación antigua), a la que voy a ir migrando la información que vaya siendo conveniente. Para ello tengo que trabajar simultáneamente con 2 bases de datos: la antigua para ir extrayendo información (mysql) y la nueva a la que añadiré o migraré información (mariadb).

¿Me podrías hacer los pasos para lo siguiente?:

1.Mantener el acceso para consulta a la base datos antigua en mysql en la parte de backend.

2.Crear un nuevo acceso para la base de datos nueva en la parte de backend.

3.Configuración de la base de datos nueva para la creación de tablas necesarias para una gestión de usuarios y permisos, similar a la que se hacía en la base de datos antigua. Hay que tener en cuenta que se debe crear un usuario inicial admin que pueda cambiar la contraseña inicial y que tenga cualquier permiso existente.

4.Creación de formulario para el acceso del usuario que lleve un control lo más seguro posible del manejo de las contraseñas en frontend.

5.Creación de formularios para la inserción, edición y eliminación de usuarios por parte de cualquier usuario con los permisos adecuados en el frontend.

6.Modificación de la página de inicio para que pueda apuntar al una página de login y una vez que acceda correctamente, a la gestión de usuarios o a un menú de opciones a realizar, entre ellas la gestión de usuarios.

Respuesta:

¡Fantástico! Es una excelente decisión. Separar la nueva lógica en una base de datos limpia mientras se consulta la antigua es una estrategia de migración profesional y muy común. Me alegra que estés listo para dar este gran paso.

Entiendo perfectamente que son muchos conceptos. No te preocupes, es normal. La mejor forma de afianzarlos es precisamente esta: construyendo funcionalidades reales paso a paso. Vamos a desglosar esta tarea, que es grande, en partes manejables.

Aquí tienes una guía completa para lograr todo lo que pides. Nos centraremos en la claridad y en los pasos lógicos.

### **Paso 1 y 2: Conectar con Dos Bases de Datos (MySQL y MariaDB)**

NestJS y TypeORM permiten gestionar múltiples conexiones a bases de datos de forma muy elegante dándoles un nombre a cada una.

#### **1a. Actualizar Docker Compose**

Añadiremos un nuevo servicio para tu base de datos MariaDB en el fichero docker-compose.yml.

**docker-compose.yml** (extracto)

YAML

version: '3.8'  
  
services:  
 # Base de datos ANTIGUA (MySQL) - Le cambiamos el nombre para más claridad  
 mysql-db-old:  
 image: mysql:8.0  
 container\_name: mysql-db-old-container  
 restart: always  
 environment:  
 MYSQL\_ROOT\_PASSWORD: 'tu\_contraseña\_de\_mysql'  
 MYSQL\_DATABASE: 'nombre\_de\_tu\_bd\_antigua'  
 ports:  
 - "3306:3306" # Puerto para la BD antigua  
 volumes:  
 - db-data-old:/var/lib/mysql  
  
 # ¡NUEVO! Base de datos NUEVA (MariaDB)  
 mariadb-db-new:  
 image: mariadb:10.6 # Usamos una imagen de MariaDB  
 container\_name: mariadb-db-new-container  
 restart: always  
 environment:  
 MYSQL\_ROOT\_PASSWORD: 'tu\_nueva\_contraseña\_segura' # Pon una contraseña segura  
 MYSQL\_DATABASE: 'gestion\_personal\_db' # Nombre de la nueva BD  
 ports:  
 - "3307:3306" # Mapeamos al puerto 3307 en tu localhost para no chocar con el 3306  
 volumes:  
 - db-data-new:/var/lib/mysql  
  
 # ... servicios backend-api y frontend-app (sin cambios por ahora)  
  
volumes:  
 db-data-old:  
 db-data-new: # Volumen para la nueva BD

#### **1b. Actualizar Variables de Entorno (.env)**

Añade las credenciales para la nueva base de datos en mi-api-gestion/.env.

**mi-api-gestion/.env**

Fragmento de código

# Base de Datos ANTIGUA (MySQL)  
DB\_OLD\_HOST=mysql-db-old  
DB\_OLD\_PORT=3306  
DB\_OLD\_USERNAME=root  
DB\_OLD\_PASSWORD=tu\_contraseña\_de\_mysql  
DB\_OLD\_DATABASE=nombre\_de\_tu\_bd\_antigua  
  
# Base de Datos NUEVA (MariaDB)  
DB\_NEW\_HOST=mariadb-db-new  
DB\_NEW\_PORT=3306 # Dentro de Docker, el puerto es el 3306 por defecto  
DB\_NEW\_USERNAME=root  
DB\_NEW\_PASSWORD=tu\_nueva\_contraseña\_segura  
DB\_NEW\_DATABASE=gestion\_personal\_db

#### **1c. Configurar Múltiples Conexiones en app.module.ts**

Este es el cambio principal. TypeOrmModule.forRootAsync aceptará un array de configuraciones.

**mi-api-gestion/src/app.module.ts**

TypeScript

import { Module } from '@nestjs/common';  
import { ConfigModule, ConfigService } from '@nestjs/config';  
import { TypeOrmModule } from '@nestjs/typeorm';  
// ... otros imports  
  
@Module({  
 imports: [  
 ConfigModule.forRoot({ isGlobal: true }),  
   
 // Array de conexiones de TypeORM  
 TypeOrmModule.forRootAsync({  
 name: 'new', // Nombre para la conexión NUEVA  
 imports: [ConfigModule],  
 inject: [ConfigService],  
 useFactory: (configService: ConfigService) => ({  
 type: 'mariadb',  
 host: configService.get<string>('DB\_NEW\_HOST'),  
 port: parseInt(configService.get<string>('DB\_NEW\_PORT'), 10),  
 username: configService.get<string>('DB\_NEW\_USERNAME'),  
 password: configService.get<string>('DB\_NEW\_PASSWORD'),  
 database: configService.get<string>('DB\_NEW\_DATABASE'),  
 entities: [\_\_dirname + '/../\*\*/\*.entity.js'], // Carga todas las entidades del proyecto  
 synchronize: true, // true en desarrollo para que TypeORM cree las tablas automáticamente  
 }),  
 }),  
  
 TypeOrmModule.forRootAsync({  
 name: 'old', // Nombre para la conexión ANTIGUA  
 imports: [ConfigModule],  
 inject: [ConfigService],  
 useFactory: (configService: ConfigService) => ({  
 type: 'mysql',  
 host: configService.get<string>('DB\_OLD\_HOST'),  
 port: parseInt(configService.get<string>('DB\_OLD\_PORT'), 10),  
 username: configService.get<string>('DB\_OLD\_USERNAME'),  
 password: configService.get<string>('DB\_OLD\_PASSWORD'),  
 database: configService.get<string>('DB\_OLD\_DATABASE'),  
 entities: [], // No queremos que maneje entidades nuevas, solo consulta  
 synchronize: false, // ¡MUY IMPORTANTE! Nunca sincronizar con la BD antigua  
 }),  
 }),  
  
 // ... otros módulos como UsersModule, etc.  
 ],  
 // ...  
})  
export class AppModule {}

**Importante:** Al inyectar repositorios, ahora tendrás que especificar el nombre de la conexión: @InjectRepository(User, 'new').

### **Paso 3: Configuración de la Nueva Base de Datos y Usuario Admin**

Vamos a crear las nuevas entidades User y Permiso y un "seeder" que creará al usuario admin la primera vez que arranque la aplicación.

#### **3a. Instalar bcrypt para contraseñas seguras**

En la terminal, dentro de mi-api-gestion:

Bash

npm install bcrypt @types/bcrypt

#### **3b. Crear Entidades para la Nueva BD**

Vamos a recrear las entidades User y Permiso para la nueva base de datos, esta vez con la lógica de contraseñas. Por claridad, puedes ubicarlas en src/database/entities/.

**mi-api-gestion/src/database/entities/user.entity.ts**

TypeScript

import { BeforeInsert, Column, Entity, JoinTable, ManyToMany, PrimaryGeneratedColumn } from "typeorm";  
import \* as bcrypt from 'bcrypt';  
import { Permiso } from "./permiso.entity";  
  
@Entity({ name: 'users' })  
export class User {  
 @PrimaryGeneratedColumn()  
 id: number;  
  
 @Column({ unique: true })  
 username: string;  
  
 @Column({ unique: true })  
 email: string;  
  
 @Column()  
 password: string;  
  
 @ManyToMany(() => Permiso, { eager: true }) // eager: true para que siempre cargue los permisos  
 @JoinTable({ name: 'users\_permisos' /\* ... \*/ })  
 permisos: Permiso[];  
  
 @BeforeInsert()  
 async hashPassword() {  
 this.password = await bcrypt.hash(this.password, 10);  
 }  
  
 async validatePassword(password: string): Promise<boolean> {  
 return bcrypt.compare(password, this.password);  
 }  
}

*Crea también la entidad Permiso de forma similar a como la hicimos antes en src/database/entities/permiso.entity.ts.*

#### **3c. Crear un "Seeder" para el usuario Admin**

Esto es un servicio que se ejecuta una vez al inicio para poblar la base de datos.

1. **Crea un DatabaseSeederModule**: nest g module database/seeder --flat
2. **Crea un DatabaseSeederService**: nest g service database/seeder --flat

**mi-api-gestion/src/database/seeder.service.ts**

TypeScript

import { Injectable, OnModuleInit } from '@nestjs/common';  
import { InjectRepository } from '@nestjs/typeorm';  
import { Repository } from 'typeorm';  
import { User } from './entities/user.entity';  
import { Permiso } from './entities/permiso.entity';  
  
@Injectable()  
export class SeederService implements OnModuleInit {  
 constructor(  
 @InjectRepository(User, 'new') private userRepository: Repository<User>,  
 @InjectRepository(Permiso, 'new') private permisoRepository: Repository<Permiso>,  
 ) {}  
  
 async onModuleInit() {  
 await this.seedPermissions();  
 await this.seedAdminUser();  
 }  
  
 async seedPermissions() {  
 const existingPermisos = await this.permisoRepository.count();  
 if (existingPermisos > 0) return;  
  
 const permisosData = [  
 { tipo: 'users:create', descripcion: 'Crear usuarios' },  
 { tipo: 'users:read', descripcion: 'Leer usuarios' },  
 { tipo: 'users:update', descripcion: 'Actualizar usuarios' },  
 { tipo: 'users:delete', descripcion: 'Eliminar usuarios' },  
 ];  
 const permisos = this.permisoRepository.create(permisosData);  
 await this.permisoRepository.save(permisos);  
 console.log('Permisos creados!');  
 }  
  
 async seedAdminUser() {  
 const existingAdmin = await this.userRepository.findOne({ where: { username: 'admin' } });  
 if (existingAdmin) return;  
   
 const allPermisos = await this.permisoRepository.find();  
  
 const admin = this.userRepository.create({  
 username: 'admin',  
 email: 'admin@example.com',  
 password: 'ChangeMe123!', // Contraseña inicial  
 permisos: allPermisos,  
 });  
 await this.userRepository.save(admin);  
 console.log('Usuario Admin creado!');  
 }  
}

*No olvides registrar las entidades y el SeederModule en tu AppModule.*

### **Paso 4: Creación de Formulario de Login Seguro**

Usaremos **JWT (JSON Web Tokens)** para la autenticación. El flujo será: el usuario envía credenciales, el backend devuelve un token, y el frontend usa ese token para las peticiones protegidas.

#### **4a. Configurar Módulo de Autenticación en el Backend**

1. **Instala las dependencias**: npm install @nestjs/passport passport passport-local @nestjs/jwt passport-jwt y sus tipos @types/....
2. **Crea un AuthModule**: nest g module auth
3. **Implementa el AuthService**: Tendrá una función validateUser (que usa bcrypt para comparar contraseñas) y una función login (que si la validación es correcta, crea y devuelve un JWT).
4. **Implementa el AuthController**: Expondrá un endpoint /auth/login.
5. **Crea una JwtStrategy**: Le enseñará a Passport cómo validar los JWT que llegan en las cabeceras de las peticiones.

#### **4b. Crear el Formulario de Login en el Frontend**

1. **Crea la página de login**: mi-app-gestion/src/app/login/page.tsx
2. **Usa una librería de formularios** como React Hook Form para gestionar el estado del formulario y la validación.
3. **Crea un Contexto de Autenticación**: Un AuthContext de React que envolverá toda la aplicación. Almacenará el estado del usuario (isAuthenticated, user, token). Cuando el login es exitoso, actualizarás este contexto.

**mi-app-gestion/src/app/login/page.tsx** (ejemplo simplificado)

TypeScript

'use client';  
import { useForm } from 'react-hook-form';  
import { Button, TextField, Container, Typography, Box } from '@mui/material';  
// ... import de tu AuthContext y servicio de API  
  
export default function LoginPage() {  
 const { register, handleSubmit } = useForm();  
 // const { login } = useAuth(); // Función de tu AuthContext  
  
 const onSubmit = async (data) => {  
 try {  
 // const userData = await api.post('/auth/login', data);  
 // login(userData); // Actualiza el contexto  
 // router.push('/dashboard');  
 } catch (error) {  
 console.error('Fallo en el login', error);  
 }  
 };  
  
 return (  
 <Container maxWidth="xs">  
 <Box component="form" onSubmit={handleSubmit(onSubmit)} sx={{ mt: 8 }}>  
 <Typography variant="h4" component="h1" gutterBottom>Login</Typography>  
 <TextField label="Username" {...register('username')} fullWidth margin="normal" />  
 <TextField label="Contraseña" type="password" {...register('password')} fullWidth margin="normal" />  
 <Button type="submit" fullWidth variant="contained" sx={{ mt: 3 }}>Acceder</Button>  
 </Box>  
 </Container>  
 );  
}

### **Paso 5: Formularios CRUD para Usuarios**

Ahora que tenemos usuarios y permisos en la nueva base de datos y un sistema de login, podemos crear la gestión de usuarios.

#### **5a. Proteger Endpoints en el Backend**

Usa los "Guards" de NestJS para proteger los endpoints de CRUD. Solo usuarios con el JWT correcto y los permisos adecuados podrán acceder.

**mi-api-gestion/src/users/users.controller.ts**

TypeScript

import { Controller, Get, UseGuards, Post, Body } from '@nestjs/common';  
import { AuthGuard } from '@nestjs/passport';  
import { HasPermission } from '../auth/decorators/has-permission.decorator'; // Un decorador personalizado  
  
@Controller('users')  
@UseGuards(AuthGuard('jwt')) // Protege todo el controlador  
export class UsersController {  
 // ...  
  
 @Post()  
 @HasPermission('users:create') // Solo usuarios con este permiso pueden crear  
 createUser(@Body() createUserDto: CreateUserDto) {  
 // ... lógica para crear usuario  
 }  
}

#### **5b. Crear Formularios CRUD en el Frontend**

En tu página de listado de usuarios (/users):

1. Añade un botón "Crear Usuario".
2. Al hacer clic, abre un **Modal de MUI (<Dialog>)**.
3. Dentro del Dialog, coloca un formulario (de nuevo, con React Hook Form) para los campos del nuevo usuario (username, email, contraseña inicial, y un selector múltiple para los permisos).
4. Para la edición, el botón "Editar" en cada fila abrirá el mismo Dialog, pero poblado con los datos del usuario a editar.
5. Usa **TanStack Query (useMutation)** para gestionar las llamadas a la API de creación, edición y eliminación. Esto te dará estados de isLoading, isSuccess, etc., y te permitirá revalidar automáticamente la lista de usuarios tras una operación exitosa.

### **Paso 6: Flujo de Navegación (Página de Inicio y Rutas Protegidas)**

#### **6a. Rutas Protegidas**

Crea un componente "wrapper" que verifique si el usuario está autenticado (usando tu AuthContext).

**mi-app-gestion/src/components/ProtectedRoute.tsx**

TypeScript

'use client';  
import { useAuth } from '@/context/AuthContext';  
import { useRouter } from 'next/navigation';  
import { useEffect } from 'react';  
  
export default function ProtectedRoute({ children }) {  
 const { isAuthenticated, isLoading } = useAuth();  
 const router = useRouter();  
  
 useEffect(() => {  
 if (!isLoading && !isAuthenticated) {  
 router.push('/login');  
 }  
 }, [isLoading, isAuthenticated, router]);  
  
 if (isLoading || !isAuthenticated) {  
 return <CircularProgress />; // O un spinner de carga  
 }  
  
 return children;  
}

Luego, en el layout de las páginas que quieres proteger (p.ej., app/dashboard/layout.tsx), envuelve el children con este componente.

#### **6b. Página de Inicio y Menú**

1. **Modifica la página de inicio (/)**: Puedes hacer que esta página compruebe el estado de autenticación. Si el usuario está logueado, redirige a /dashboard. Si no, redirige a /login.
2. **Crea una página /dashboard**: Esta será la página principal después del login.
3. **Crea un Layout principal con menú**: Crea un layout (app/dashboard/layout.tsx) que tenga una barra de navegación lateral o superior persistente (usando <AppBar> y <Drawer> de MUI). Este menú contendrá los enlaces a las diferentes secciones, como "Gestión de Usuarios".

Este es un plan de acción muy completo. Te recomiendo que vayas implementando cada paso con calma. Empieza por la configuración de las dos bases de datos, verifica que puedes conectar a ambas y luego avanza hacia la autenticación y el CRUD. ¡Mucho ánimo, estás construyendo una aplicación muy robusta!