Pregunta:

¿Puedes detallarme el punto 5 sobre creación de formularios para inserción, edición y eliminación de usuarios por parte de cualquier usuario con los permisos adecuados en el frontend y la creación del CRUD necesario en el backend?

Respuesta:

¡Excelente! Detallar el punto 5 es crucial. Aquí es donde tu aplicación pasa de ser una herramienta de visualización a ser una plataforma de gestión interactiva y segura. Construiremos un sistema CRUD (Create, Read, Update, Delete) completo, protegiendo cada acción con el sistema de permisos que diseñamos.

Este proceso es extenso, pero lo dividiremos en pasos lógicos y manejables, cubriendo primero el backend y luego el frontend.

### **Parte 1: Backend (NestJS) - Construyendo el API para CRUD**

Ampliaremos nuestro UsersModule para que maneje la creación, actualización y eliminación de usuarios, protegiendo cada endpoint con "Guards" de permisos.

#### **Paso 5.1: Crear DTOs (Data Transfer Objects) para Validación**

Los DTOs son objetos que definen la forma de los datos que llegan a nuestros endpoints. Nos permiten validar automáticamente la información entrante.

1. **Instala las dependencias de validación:**  
   Bash  
   npm install class-validator class-transformer
2. **Activa el ValidationPipe globalmente** en mi-api-gestion/src/main.ts. Esto hará que NestJS valide automáticamente todos los DTOs que lleguen a los controladores.  
   TypeScript  
   // mi-api-gestion/src/main.ts  
   import { ValidationPipe } from '@nestjs/common';  
   // ...  
   async function bootstrap() {  
    const app = await NestFactory.create(AppModule);  
     
    app.useGlobalPipes(new ValidationPipe({  
    whitelist: true, // Elimina propiedades que no están en el DTO  
    forbidNonWhitelisted: true, // Lanza un error si se envían propiedades no permitidas  
    transform: true, // Transforma los payloads a instancias de DTO  
    }));  
     
    // ... resto de la configuración (CORS, Swagger, etc.)  
    await app.listen(3001);  
   }  
   bootstrap();
3. **Crea los DTOs** dentro de una nueva carpeta src/users/dto.  
   **mi-api-gestion/src/users/dto/create-user.dto.ts**  
   TypeScript  
   import { IsEmail, IsNotEmpty, IsString, MinLength, IsArray, IsInt } from 'class-validator';  
     
   export class CreateUserDto {  
    @IsString()  
    @IsNotEmpty()  
    username: string;  
     
    @IsEmail()  
    email: string;  
     
    @IsString()  
    @MinLength(8, { message: 'La contraseña debe tener al menos 8 caracteres' })  
    password: string;  
     
    @IsArray()  
    @IsInt({ each: true }) // Valida que cada elemento del array sea un entero  
    permisoIds: number[];  
   }  
     
   **mi-api-gestion/src/users/dto/update-user.dto.ts**  
   TypeScript  
   import { IsEmail, IsOptional, IsString, MinLength, IsArray, IsInt } from 'class-validator';  
     
   export class UpdateUserDto {  
    @IsOptional()  
    @IsString()  
    username?: string;  
     
    @IsOptional()  
    @IsEmail()  
    email?: string;  
     
    @IsOptional()  
    @IsString()  
    @MinLength(8, { message: 'La contraseña debe tener al menos 8 caracteres' })  
    password?: string;  
     
    @IsOptional()  
    @IsArray()  
    @IsInt({ each: true })  
    permisoIds?: number[];  
   }

#### **Paso 5.2: Crear un Guard de Permisos Personalizado**

Este Guard verificará si el usuario logueado (cuyos datos están en el token JWT) tiene los permisos necesarios para realizar una acción.

1. **Crea el Decorador** para asignar permisos a las rutas.  
   **mi-api-gestion/src/auth/decorators/has-permission.decorator.ts**  
   TypeScript  
   import { SetMetadata } from '@nestjs/common';  
   export const PERMISSION\_KEY = 'permission';  
   export const HasPermission = (permission: string) => SetMetadata(PERMISSION\_KEY, permission);
2. **Crea el PermissionsGuard.**  
   **mi-api-gestion/src/auth/guards/permissions.guard.ts**  
   TypeScript  
   import { Injectable, CanActivate, ExecutionContext } from '@nestjs/common';  
   import { Reflector } from '@nestjs/core';  
   import { PERMISSION\_KEY } from '../decorators/has-permission.decorator';  
     
   @Injectable()  
   export class PermissionsGuard implements CanActivate {  
    constructor(private reflector: Reflector) {}  
     
    canActivate(context: ExecutionContext): boolean {  
    const requiredPermission = this.reflector.get<string>(  
    PERMISSION\_KEY,  
    context.getHandler(),  
    );  
    if (!requiredPermission) {  
    return true; // Si no se especifica ningún permiso, se permite el acceso  
    }  
    const { user } = context.switchToHttp().getRequest();  
    // user.permisos viene del payload del token que definimos en JwtStrategy  
    return user.permisos?.includes(requiredPermission);  
    }  
   }

#### **Paso 5.3: Expandir UsersService con la Lógica CRUD**

Añade los métodos para crear, actualizar y borrar en mi-api-gestion/src/users/users.service.ts.

TypeScript

// En UsersService  
import { In } from 'typeorm';  
// ...  
  
@Injectable()  
export class UsersService {  
 constructor(  
 @InjectRepository(User, 'new') private userRepository: Repository<User>,  
 @InjectRepository(Permiso, 'new') private permisoRepository: Repository<Permiso>,  
 ) {}  
  
 // ... findAll y findOneByUsername sin cambios ...  
  
 async create(createUserDto: CreateUserDto): Promise<User> {  
 const { permisoIds, ...userData } = createUserDto;  
 const permisos = await this.permisoRepository.findBy({ id: In(permisoIds) });  
   
 const newUser = this.userRepository.create({  
 ...userData,  
 permisos,  
 });  
  
 return this.userRepository.save(newUser);  
 }  
  
 async update(id: number, updateUserDto: UpdateUserDto): Promise<User> {  
 const { permisoIds, ...userData } = updateUserDto;  
 const user = await this.userRepository.findOneBy({ id });  
 if (!user) {  
 throw new NotFoundException(`Usuario con ID ${id} no encontrado`);  
 }  
  
 if (permisoIds) {  
 user.permisos = await this.permisoRepository.findBy({ id: In(permisoIds) });  
 }  
   
 // Si se envía una nueva contraseña, la hasheamos antes de guardar  
 if (userData.password) {  
 userData.password = await bcrypt.hash(userData.password, 10);  
 }  
  
 Object.assign(user, userData);  
 return this.userRepository.save(user);  
 }  
  
 async remove(id: number): Promise<void> {  
 const result = await this.userRepository.delete(id);  
 if (result.affected === 0) {  
 throw new NotFoundException(`Usuario con ID ${id} no encontrado`);  
 }  
 }  
}

*(Nota: Asegúrate de inyectar también el PermisoRepository)*

#### **Paso 5.4: Expandir UsersController con los Endpoints CRUD Protegidos**

Ahora, junta todo en el controlador.

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

TypeScript

import { Controller, Get, Post, Body, Patch, Param, Delete, UseGuards, ParseIntPipe } from '@nestjs/common';  
import { AuthGuard } from '@nestjs/passport';  
import { PermissionsGuard } from '../auth/guards/permissions.guard';  
import { HasPermission } from '../auth/decorators/has-permission.decorator';  
// ...  
  
@Controller('users')  
@UseGuards(AuthGuard('jwt'), PermissionsGuard) // Protege todo el controlador  
export class UsersController {  
 constructor(private readonly usersService: UsersService) {}  
  
 @Get()  
 @HasPermission('users:read')  
 findAll() {  
 return this.usersService.findAll();  
 }  
   
 @Post()  
 @HasPermission('users:create')  
 create(@Body() createUserDto: CreateUserDto) {  
 return this.usersService.create(createUserDto);  
 }  
  
 @Patch(':id')  
 @HasPermission('users:update')  
 update(@Param('id', ParseIntPipe) id: number, @Body() updateUserDto: UpdateUserDto) {  
 return this.usersService.update(id, updateUserDto);  
 }  
  
 @Delete(':id')  
 @HasPermission('users:delete')  
 remove(@Param('id', ParseIntPipe) id: number) {  
 return this.usersService.remove(id);  
 }  
}

### **Parte 2: Frontend (Next.js/MUI) - La Interfaz de Gestión**

Ahora construiremos la interfaz para interactuar con nuestro nuevo API CRUD.

#### **Paso 5.5: Crear un Hook de Permisos (usePermissions)**

Para mantener el código limpio, crea un hook personalizado que facilite la comprobación de permisos.

**mi-app-gestion/src/hooks/usePermissions.ts**

TypeScript

import { useAuth } from "@/context/AuthContext";  
  
export const usePermissions = (requiredPermission: string): boolean => {  
 const { user } = useAuth();  
 if (!user || !user.permisos) {  
 return false;  
 }  
 return user.permisos.includes(requiredPermission);  
};

#### **Paso 5.6: Modificar la Página de Usuarios para el CRUD**

Aquí es donde ocurre la mayor parte de la magia en el frontend. Usaremos un Dialog de MUI para el formulario y TanStack Query para las mutaciones.

**mi-app-gestion/src/app/users/page.tsx** (extracto de los cambios y adiciones)

TypeScript

'use client';  
  
import { useState } from 'react';  
import { useQuery, useMutation, useQueryClient } from '@tanstack/react-query';  
import { Button, IconButton, Dialog, DialogTitle, DialogContent, Box, Alert } from '@mui/material';  
import AddIcon from '@mui/icons-material/Add';  
import EditIcon from '@mui/icons-material/Edit';  
import DeleteIcon from '@mui/icons-material/Delete';  
  
import { User } from '@/types/user';  
import api from '@/lib/api';  
import { usePermissions } from '@/hooks/usePermissions';  
import UserForm from '@/components/UserForm'; // Un nuevo componente que crearemos  
import ConfirmationDialog from '@/components/ConfirmationDialog'; // Otro nuevo componente  
  
// ...  
  
export default function UsersListPage() {  
 const queryClient = useQueryClient();  
 const [isFormOpen, setIsFormOpen] = useState(false);  
 const [isConfirmOpen, setIsConfirmOpen] = useState(false);  
 const [editingUser, setEditingUser] = useState<User | null>(null);  
 const [deletingUserId, setDeletingUserId] = useState<number | null>(null);  
  
 // Permisos  
 const canRead = usePermissions('users:read');  
 const canCreate = usePermissions('users:create');  
 const canUpdate = usePermissions('users:update');  
 const canDelete = usePermissions('users:delete');  
  
 // Query para obtener usuarios (como antes)  
 const { data: users, isLoading, isError } = useQuery<User[]>({  
 queryKey: ['users'],  
 queryFn: () => api.get('/users').then(res => res.data),  
 enabled: canRead, // Solo ejecuta la query si tiene permiso de lectura  
 });  
  
 // Mutación para crear/actualizar usuario  
 const userMutation = useMutation({  
 mutationFn: (userData: any) =>   
 editingUser   
 ? api.patch(`/users/${editingUser.id}`, userData)   
 : api.post('/users', userData),  
 onSuccess: () => {  
 queryClient.invalidateQueries({ queryKey: ['users'] });  
 setIsFormOpen(false);  
 },  
 });  
  
 // Mutación para eliminar usuario  
 const deleteMutation = useMutation({  
 mutationFn: (id: number) => api.delete(`/users/${id}`),  
 onSuccess: () => {  
 queryClient.invalidateQueries({ queryKey: ['users'] });  
 setIsConfirmOpen(false);  
 },  
 });  
  
 // Handlers para abrir/cerrar dialogs  
 const handleOpenCreateForm = () => {  
 setEditingUser(null);  
 setIsFormOpen(true);  
 };  
   
 const handleOpenEditForm = (user: User) => {  
 setEditingUser(user);  
 setIsFormOpen(true);  
 };  
  
 const handleOpenDeleteConfirm = (id: number) => {  
 setDeletingUserId(id);  
 setIsConfirmOpen(true);  
 };  
  
 const handleFormSubmit = (data: any) => {  
 userMutation.mutate(data);  
 };  
   
 const handleDeleteConfirm = () => {  
 if (deletingUserId) {  
 deleteMutation.mutate(deletingUserId);  
 }  
 };  
  
 if (!canRead) return <Alert severity="error">No tienes permiso para ver esta página.</Alert>;  
  
 // ... el resto de tu JSX (Container, Table, etc.)  
   
 // Dentro de tu JSX:  
 // Botón para crear  
 <Box>  
 {canCreate && <Button startIcon={<AddIcon />} onClick={handleOpenCreateForm}>Crear Usuario</Button>}  
 </Box>  
  
 // Columna de acciones en la tabla  
 <TableCell>  
 {canUpdate && <IconButton onClick={() => handleOpenEditForm(user)}><EditIcon /></IconButton>}  
 {canDelete && <IconButton onClick={() => handleOpenDeleteConfirm(user.id)}><DeleteIcon /></IconButton>}  
 </TableCell>  
  
 // Dialogs al final del componente  
 <Dialog open={isFormOpen} onClose={() => setIsFormOpen(false)}>  
 <DialogTitle>{editingUser ? 'Editar Usuario' : 'Crear Usuario'}</DialogTitle>  
 <DialogContent>  
 <UserForm   
 onSubmit={handleFormSubmit}   
 initialData={editingUser}  
 isSubmitting={userMutation.isPending}   
 />  
 </DialogContent>  
 </Dialog>  
   
 <ConfirmationDialog  
 open={isConfirmOpen}  
 onClose={() => setIsConfirmOpen(false)}  
 onConfirm={handleDeleteConfirm}  
 title="Confirmar Eliminación"  
 description="¿Estás seguro de que quieres eliminar este usuario? Esta acción no se puede deshacer."  
 isSubmitting={deleteMutation.isPending}  
 />  
}

#### **Paso 5.7: Crear el Componente de Formulario UserForm**

Este componente reutilizable contendrá el formulario en sí.

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

TypeScript

import { useForm } from 'react-hook-form';  
import { TextField, Button, Box, Select, MenuItem, FormControl, InputLabel, Chip } from '@mui/material';  
// ... y más imports  
  
// Este componente recibe la función de submit y los datos iniciales  
const UserForm = ({ onSubmit, initialData, isSubmitting }) => {  
 const { register, handleSubmit, control } = useForm({  
 defaultValues: initialData || { username: '', email: '', password: '', permisoIds: [] },  
 });  
   
 // Aquí harías una query para obtener todos los permisos disponibles y mostrarlos en el select  
 // const { data: allPermissions } = useQuery(['permissions'], ...);  
  
 return (  
 <Box component="form" onSubmit={handleSubmit(onSubmit)}>  
 <TextField label="Username" {...register('username')} fullWidth margin="normal" />  
 <TextField label="Email" {...register('email')} fullWidth margin="normal" />  
 <TextField label="Contraseña (dejar en blanco para no cambiar)" type="password" {...register('password')} fullWidth margin="normal" />  
 {/\* Aquí iría un <Select> múltiple para los permisos \*/}  
 <Button type="submit" disabled={isSubmitting}>Guardar</Button>  
 </Box>  
 );  
};  
export default UserForm;

*(El formulario se ha simplificado, pero te da la estructura completa para implementarlo con react-hook-form y MUI).*

Con estos pasos, has construido un sistema CRUD completo, seguro y reactivo. Cada acción está protegida por permisos en el backend, y el frontend responde dinámicamente, mostrando u ocultando acciones y actualizando la vista automáticamente tras cada operación.