Pregunta:

No he usado nunca jestjs con typeorm, ya tengo conectada la base de datos con mi api de nest y esta, a su vez, con mi aplicación. ¿Podrías darme un ejemplo completo para acceder a la tabla de usuarios de la base de datos con la definición que te paso más abajo, el acceso con nest mediante typeorm y su exposición en como lista de usuarios en la aplicación que tiene el framework next.js?

La estructura de la tabla de usuarios definida en SQL es:

`

CREATE TABLE IF NOT EXISTS `users` (

`id` INT(11) NOT NULL AUTO\_INCREMENT,

`username` VARCHAR(50) CHARACTER SET 'utf8' COLLATE 'utf8\_bin' NOT NULL,

`password` VARCHAR(255) CHARACTER SET 'utf8' COLLATE 'utf8\_bin' NOT NULL,

`email` VARCHAR(100) CHARACTER SET 'utf8' COLLATE 'utf8\_bin' NOT NULL,

`activated` TINYINT(1) NOT NULL DEFAULT '1',

`banned` TINYINT(1) NOT NULL DEFAULT '0',

`ban\_reason` VARCHAR(255) CHARACTER SET 'utf8' COLLATE 'utf8\_bin' NULL DEFAULT NULL,

`new\_password\_key` VARCHAR(50) CHARACTER SET 'utf8' COLLATE 'utf8\_bin' NULL DEFAULT NULL,

`new\_password\_requested` DATETIME NULL DEFAULT NULL,

`new\_email` VARCHAR(100) CHARACTER SET 'utf8' COLLATE 'utf8\_bin' NULL DEFAULT NULL,

`new\_email\_key` VARCHAR(50) CHARACTER SET 'utf8' COLLATE 'utf8\_bin' NULL DEFAULT NULL,

`last\_ip` VARCHAR(40) CHARACTER SET 'utf8' COLLATE 'utf8\_bin' NOT NULL,

`last\_login` DATETIME NOT NULL DEFAULT '0000-00-00 00:00:00',

`created` DATETIME NOT NULL DEFAULT '0000-00-00 00:00:00',

`modified` TIMESTAMP NOT NULL DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

PRIMARY KEY (`id`))

ENGINE = MyISAM

DEFAULT CHARACTER SET = utf8

COLLATE = utf8\_bin;

`

Respuesta:

¡Por supuesto! Es una excelente idea poner en práctica todo el flujo de trabajo, desde la base de datos hasta la interfaz de usuario. El ejemplo que pides es el "Hola Mundo" de cualquier aplicación full-stack y te servirá de base para toda la funcionalidad futura.

A pesar de que mencionaste jestjs en tu pregunta, el núcleo de tu solicitud es sobre cómo definir, exponer y consumir los datos. Nos centraremos en ese flujo completo. La configuración de tests con Jest para TypeORM es un tema más avanzado que podemos abordar después, pero esta estructura es el prerrequisito indispensable para poder testearla.

Aquí tienes la guía completa y detallada.

### **Parte 1: Backend (NestJS, TypeORM)**

Vamos a crear todo lo necesario en la API para leer la tabla users, manejar los datos de forma segura y exponerlos en un endpoint.

#### **Paso 1.1: Crear la Entidad User con TypeORM**

Esta clase es la representación de tu tabla SQL en código TypeScript. NestJS la usará para entender cómo interactuar con la tabla users.

1. Primero, vamos a organizar el código. Crea una nueva carpeta users dentro de mi-api-gestion/src. Dentro de users, crea otra carpeta llamada entities.
2. Dentro de src/users/entities/, crea un fichero llamado user.entity.ts.

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

TypeScript

import {  
 Entity,  
 PrimaryGeneratedColumn,  
 Column,  
 CreateDateColumn,  
 UpdateDateColumn,  
} from 'typeorm';  
  
@Entity({ name: 'users' }) // Mapea a la tabla 'users'  
export class User {  
 @PrimaryGeneratedColumn()  
 id: number;  
  
 @Column({ type: 'varchar', length: 50, unique: true })  
 username: string;  
  
 // IMPORTANTE: Nunca expondremos este campo en la API.  
 @Column({ type: 'varchar', length: 255, select: false }) // 'select: false' evita que se obtenga por defecto  
 password: string;  
  
 @Column({ type: 'varchar', length: 100, unique: true })  
 email: string;  
  
 @Column({ type: 'tinyint', width: 1, default: 1 })  
 activated: boolean;  
  
 @Column({ type: 'tinyint', width: 1, default: 0 })  
 banned: boolean;  
  
 @Column({ type: 'varchar', length: 255, nullable: true })  
 ban\_reason: string;  
   
 // Omitimos los campos de reseteo de contraseña/email por simplicidad en este ejemplo  
 // pero se añadirían como @Column si fueran necesarios.  
 // new\_password\_key, new\_password\_requested, new\_email, new\_email\_key...  
  
 @Column({ type: 'varchar', length: 40 })  
 last\_ip: string;  
  
 @Column({ type: 'datetime', default: '0000-00-00 00:00:00' })  
 last\_login: Date;  
  
 // TypeORM puede gestionar automáticamente las fechas de creación y modificación  
 // Si la tabla ya lo hace, esto se alinea perfectamente.  
 @Column({ type: 'datetime', default: '0000-00-00 00:00:00' })  
 created: Date;  
  
 @UpdateDateColumn() // Gestiona automáticamente la columna `modified`  
 modified: Date;  
}

#### **Paso 1.2: Generar el Módulo, Controlador y Servicio**

Usaremos la CLI de NestJS para crear la estructura básica para la gestión de usuarios. Abre una terminal en el directorio mi-api-gestion y ejecuta:

Bash

# Genera el módulo que agrupará toda la lógica de usuarios  
nest generate module users  
  
# Genera el controlador que manejará las peticiones HTTP  
nest generate controller users --no-spec  
  
# Genera el servicio que contendrá la lógica de negocio  
nest generate service users --no-spec

#### **Paso 1.3: Implementar el Servicio UsersService**

El servicio se encargará de comunicarse con la base de datos a través del Repositorio de TypeORM.

Edita el fichero mi-api-gestion/src/users/users.service.ts:

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

TypeScript

import { Injectable } from '@nestjs/common';  
import { InjectRepository } from '@nestjs/typeorm';  
import { Repository } from 'typeorm';  
import { User } from './entities/user.entity';  
  
@Injectable()  
export class UsersService {  
 constructor(  
 @InjectRepository(User)  
 private usersRepository: Repository<User>,  
 ) {}  
  
 /\*\*  
 \* Encuentra todos los usuarios y selecciona campos específicos para no exponer datos sensibles.  
 \*/  
 async findAll(): Promise<User[]> {  
 // Usamos la opción 'select' para explícitamente pedir los campos que queremos.  
 // Es una capa extra de seguridad para NUNCA devolver hashes de contraseñas.  
 return this.usersRepository.find({  
 select: {  
 id: true,  
 username: true,  
 email: true,  
 activated: true,  
 banned: true,  
 last\_login: true,  
 created: true,  
 modified: true,  
 },  
 order: {  
 id: 'ASC',  
 },  
 });  
 }  
  
 // Aquí añadirías otros métodos como findOne, create, update, delete...  
}

#### **Paso 1.4: Implementar el Controlador UsersController**

El controlador define el endpoint (/users) y lo conecta con el método del servicio.

Edita el fichero mi-api-gestion/src/users/users.controller.ts:

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

TypeScript

import { Controller, Get } from '@nestjs/common';  
import { UsersService } from './users.service';  
import { ApiOperation, ApiResponse, ApiTags } from '@nestjs/swagger';  
import { User } from './entities/user.entity';  
  
@ApiTags('Users') // Agrupa los endpoints en Swagger bajo la etiqueta 'Users'  
@Controller('users')  
export class UsersController {  
 constructor(private readonly usersService: UsersService) {}  
  
 @Get()  
 @ApiOperation({ summary: 'Obtener una lista de todos los usuarios' })  
 @ApiResponse({ status: 200, description: 'Lista de usuarios recuperada con éxito.', type: [User] })  
 findAll() {  
 return this.usersService.findAll();  
 }  
}

#### **Paso 1.5: Registrar el Módulo y la Entidad**

Finalmente, tenemos que decirle a NestJS que el UsersModule existe y que debe reconocer la entidad User.

1. Primero, en el UsersModule, importa TypeOrmModule.  
   **mi-api-gestion/src/users/users.module.ts**  
   TypeScript  
   import { Module } from '@nestjs/common';  
   import { TypeOrmModule } from '@nestjs/typeorm';  
   import { UsersService } from './users.service';  
   import { UsersController } from './users.controller';  
   import { User } from './entities/user.entity';  
     
   @Module({  
    imports: [TypeOrmModule.forFeature([User])], // Registra la entidad User en este módulo  
    controllers: [UsersController],  
    providers: [UsersService],  
   })  
   export class UsersModule {}
2. Ahora, importa el UsersModule en el módulo principal de la aplicación.  
   **mi-api-gestion/src/app.module.ts** (extracto)  
   TypeScript  
   // ... otras importaciones  
   import { UsersModule } from './users/users.module';  
     
   @Module({  
    imports: [  
    // ... ConfigModule y TypeOrmModule.forRootAsync como lo tenías antes  
    UsersModule, // ¡Añade esta línea!  
    ],  
    controllers: [AppController],  
    providers: [AppService],  
   })  
   export class AppModule {}

¡El backend está listo! Si reinicias la API, ahora tendrás un endpoint GET http://localhost:3001/users que devuelve una lista de usuarios en formato JSON.

### **Parte 2: Frontend (Next.js, TanStack Query)**

Ahora vamos a consumir ese endpoint y mostrar los datos en nuestra aplicación de Next.js.

#### **Paso 2.1: Definir el Tipo User en el Frontend**

Es una buena práctica tener una definición de tipos en el frontend que coincida con lo que la API devuelve. **Importante:** este tipo no debe incluir password ni otros campos sensibles.

Crea un fichero en mi-app-gestion/src/types/user.ts:

**mi-app-gestion/src/types/user.ts**

TypeScript

export interface User {  
 id: number;  
 username: string;  
 email: string;  
 activated: boolean;  
 banned: boolean;  
 last\_login: string; // Las fechas suelen llegar como strings en JSON  
 created: string;  
 modified: string;  
}

#### **Paso 2.2: Crear la Página de la Lista de Usuarios**

Usaremos el App Router de Next.js para crear una nueva ruta en /users.

1. Crea una nueva carpeta users dentro de mi-app-gestion/src/app.
2. Dentro de src/app/users, crea un fichero llamado page.tsx.

**mi-app-gestion/src/app/users/page.tsx**

TypeScript

'use client'; // Este componente necesita ser de cliente para usar hooks  
  
import { useQuery } from '@tanstack/react-query';  
import api from '@/lib/api'; // Nuestra instancia de Axios configurada  
import { User } from '@/types/user';  
  
// 1. Función para obtener los datos de la API  
const fetchUsers = async (): Promise<User[]> => {  
 const { data } = await api.get('/users');  
 return data;  
};  
  
// 2. El componente de la página  
export default function UsersListPage() {  
 const { data: users, isLoading, isError, error } = useQuery<User[], Error>({  
 queryKey: ['users'], // Clave única para esta consulta en la caché  
 queryFn: fetchUsers, // Función que se ejecutará para obtener los datos  
 });  
  
 // 3. Manejo de estados de carga y error  
 if (isLoading) {  
 return (  
 <div className="flex justify-center items-center h-screen">  
 <p className="text-xl">Cargando usuarios...</p>  
 </div>  
 );  
 }  
  
 if (isError) {  
 return (  
 <div className="flex justify-center items-center h-screen text-red-500">  
 <p>Error al cargar los usuarios: {error.message}</p>  
 </div>  
 );  
 }  
  
 // 4. Renderizado de la lista de usuarios  
 return (  
 <div className="container mx-auto p-8">  
 <h1 className="text-3xl font-bold mb-6">Lista de Usuarios</h1>  
 <div className="bg-white shadow-md rounded-lg overflow-hidden">  
 <table className="min-w-full leading-normal">  
 <thead>  
 <tr className="bg-gray-200 text-gray-600 uppercase text-sm leading-normal">  
 <th className="py-3 px-6 text-left">ID</th>  
 <th className="py-3 px-6 text-left">Username</th>  
 <th className="py-3 px-6 text-left">Email</th>  
 <th className="py-3 px-6 text-center">Estado</th>  
 </tr>  
 </thead>  
 <tbody className="text-gray-600 text-sm font-light">  
 {users?.map((user) => (  
 <tr key={user.id} className="border-b border-gray-200 hover:bg-gray-100">  
 <td className="py-3 px-6 text-left whitespace-nowrap">{user.id}</td>  
 <td className="py-3 px-6 text-left">{user.username}</td>  
 <td className="py-3 px-6 text-left">{user.email}</td>  
 <td className="py-3 px-6 text-center">  
 {user.banned ? (  
 <span className="bg-red-200 text-red-600 py-1 px-3 rounded-full text-xs">Baneado</span>  
 ) : user.activated ? (  
 <span className="bg-green-200 text-green-600 py-1 px-3 rounded-full text-xs">Activo</span>  
 ) : (  
 <span className="bg-yellow-200 text-yellow-600 py-1 px-3 rounded-full text-xs">Inactivo</span>  
 )}  
 </td>  
 </tr>  
 ))}  
 </tbody>  
 </table>  
 </div>  
 </div>  
 );  
}

### **Poniéndolo Todo en Marcha**

1. Asegúrate de que no tienes ninguna instancia de la API o del frontend corriendo en otras terminales.
2. Abre una terminal en la raíz de tu proyecto (mi-proyecto-gestion/).
3. Levanta todo el entorno con Docker Compose:  
   Bash  
   docker-compose up --build
4. Espera a que todos los servicios se inicien.
5. Abre tu navegador y ve a **http://localhost:3000/users**.

Deberías ver una tabla con la lista de usuarios extraída directamente de tu base de datos MySQL, pasando por tu API de NestJS y renderizada en tu aplicación Next.js. ¡Has completado el ciclo completo!