

Bueno voy a empezar con el desarrollo y lo primero que quiero entender a toda costa es lo que hay ya. He dedicado casi dos días a entender typescript así que ahora vengo mejor preparado. Veo varios archivos te los voy a mandar y te digo lo que entiendo y mis posibles dudas, así a primer vistazo. Seguramente me equivocaré bastante pero para mí esto es toma de contacto, tenme paciencia please.

En primer lugar tengo app.controller.ts

```
import { Controller, Get } from '@nestjs/common';
import { AppService } from './app.service';
@Controller()
export class AppController {
  constructor(private readonly appService: AppService) {}
  @Get() getHello(): string {
    return this.appService.getHello();
  }
}
```

Para empezar usa los imports. Se usan como una especie de desestructuraciones similar a como se hace en react. Luego se usan decoradores similares a como se hace en ANGULAR. los decoradores sirven para configurar el código que sigue y hacer que el entorno lo entienda de cierta forma. En este caso Controller indica que eso es un controlador y get que es una petición get. appService me imagino que será una app de prueba para mostrar el hello. En cuanto al constructor se pasa el appService para iniciar la clase con el appService. Se que esto forma parte de eso que llaman inyección de dependencias, pero no sé exactamente donde se lleva a cabo esa instanciación.

import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
@Module({
 imports: [],
 controllers: [AppController],
 providers: [AppService],
})
export class AppModule {}

Esto es app.module.ts Vendría a ser el contenedor donde todo lo que pertenece al grupo de lo mismo se agrupa. Por ejemplo, el controlador el servicio y lo que sea que pertenezca al mismo grupo de cosas, se agrupa en el module para indicar que forma parte de ese contenedor. En este tipo de entornos entiendo que todo se basa en agrupar en módulos.

```
import { Injectable } from '@nestjs/common';
@Injectable()
export class AppService {
  getHello(): string {
    return 'Hello World!';
  }
}
```

Siguiendo con el service, veo que hay un método que retorna un hello, pero no entiendo muy bien por qué se exporta la clase con export ni qué es el decorador Injectable..

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.listen(process.env.PORT ?? 3000);
}
bootstrap();
```

Este sería el main.ts Entiendo que aquí se importa la librería nestjs y se construye la aplicación como aplicación de nestjs llamando al método de creación de dicha librería, además se configura la aplicación para que escuche un puerto determinado. Entiendo que hay más cosas que se pueden tocar aunque por defecto viene así.

```
import { Test, TestingModule } from '@nestjs/testing';
import { AppController } from './app.controller';
import { AppService } from './app.service';
describe('AppController', () => {
  let appController: AppController;
  beforeEach(async () => {
    const app: TestingModule = await Test.createTestingModule({
      controllers: [AppController],
      providers: [AppService],
    }).compile();
    appController = app.get<AppController>(AppController);
  });
  describe('root', () => {
    it('should return "Hello World!"', () => {
      expect(appController.getHello()).toBe('Hello World!');
    });
  });
});
```

Finalmente aquí estaría el app.controller.spec.ts. Me llama la atención que en visual studio el icono se ve de color rojo (lo que son las letras de TS). Sinceramente parece una especie de cargador de pruebas, pero no logro a alcanzar a entender como funciona. Creo que es un nuevo mundo para mí. Bueno qué opinas. La verdad es que cuando vine sin saber nada de typescript era incluso más mareante. Ahora que se como se usa el tipado veo el código con otros ojos y es como que se ver lo importante y

entenderlo todo mejor de un vistazo. Si quieres puedes decirme que te parece mi primer vistazo y ayudarme a entenderlo mejor.

Análisis de tu comprensión inicial de NestJS

Vas por muy buen camino, ¡felicitaciones! Tu análisis demuestra que ya estás entendiendo los conceptos fundamentales. Vamos a profundizar y corregir algunos detalles:

☐ Tu comprensión es correcta en:

1. **Decoradores:** Efectivamente son como en Angular, metadata que configura clases/métodos.
2. **Módulos:** Son contenedores lógicos que agrupan funcionalidad relacionada.
3. **Estructura básica:** Entiendes el flujo entre controller → service → module.
4. **main.ts:** Correcto, es el punto de entrada que crea la aplicación NestJS.

☐ Puntos para aclarar:

1. Sobre `@Injectable()` y la Inyección de Dependencias

```
typescript
@Injectable()
export class AppService {
  getHello(): string {
    return 'Hello World!';
  }
}
```

- **`@Injectable()`:** Marca la clase como "proveedora" que puede ser inyectada en otros lugares.
- **Inyección:** NestJS crea automáticamente las instancias (en tiempo de ejecución) mediante su **sistema de inyección de dependencias**.
- **Dónde se instancia:** En el `AppModule`, cuando declaras `providers:` `[AppService]`, NestJS registra esta clase para su inyección.

2. Exportación de clases (`export`)

```
typescript
export class AppService {...}
```

- El `export` es de TypeScript/ES6, no específico de NestJS.
- Permite que la clase sea importada en otros archivos:

```
typescript
```

- `import { AppService } from './app.service';`

3. Archivo de pruebas (app.controller.spec.ts)

typescript

```
describe('AppController', () => {
  let appController: AppController;

  beforeEach(async () => {
    const app: TestingModule = await Test.createTestingModule({
      controllers: [AppController],
      providers: [AppService],
    }).compile();

    appController = app.get<AppController>(AppController);
  });

  it('should return "Hello World!"', () => {
    expect(appController.getHello()).toBe('Hello World!');
  });
});
```

- Es un test unitario usando **Jest** (framework de testing que viene con NestJS).
- describe/it: Bloques para organizar tests.
- beforeEach: Configuración antes de cada test.
- createTestingModule: Crea un módulo NestJS "fake" solo para testing.
- El icono rojo en VS Code es normal para archivos .spec.ts.

□ Diagrama de flujo completo

text

```
main.ts → Crea AppModule → Declara:
    └─ AppController (rutas HTTP)
    └─ AppService (lógica)
        ↑
    Inyección de Dependencias
```

□ □ Consejos para tu desarrollo

1. Jerarquía típica:

bash

- src/
 feature/
 └─ feature.controller.ts # Maneja rutas HTTP
 └─ feature.service.ts # Contiene lógica de negocio
 └─ feature.module.ts # Agrupa todo
 └─ dto/ # Objetos para transferencia de datos

• Ejemplo de Controller más completo:

typescript

```
2. @Controller('tasks') // Prefijo de ruta: /tasks
3. export class TasksController {
4.   constructor(private tasksService: TasksService) {}
```

```
5.
6.   @Get() // GET /tasks
7.   findAll() {
8.       return this.tasksService.findAll();
9.   }
10.
11.   @Post() // POST /tasks
12.   create(@Body() createTaskDto: CreateTaskDto) {
13.       return this.tasksService.create(createTaskDto);
14.   }
15. }
```

16. **Variables de entorno:** Usa @nestjs/config para manejar .env.

□ Próximos pasos recomendados

1. Crea tu primer módulo:

```
bash
```

```
• nest generate module tasks
nest generate controller tasks
nest generate service tasks
```

• Conecta Prisma:

```
bash
```

- ```
2. npm install prisma @prisma/client
3. npx prisma init
```
4. **Experimenta con:**
- DTOs con validación (class-validator)
  - Manejo de errores personalizados
  - Documentación con Swagger

¿Quieres que profundicemos en algún aspecto en particular? ¡Tu base ya es muy sólida para comenzar a desarrollar! □

Bueno, ahora quiero hacer un CRUD de usuarios, imagino que tengo que crear una carpeta para usuarios y crear todo ahí?

## Estructura para CRUD de Usuarios en NestJS

Sí, exactamente. Vamos a crear una estructura modular para el CRUD de usuarios. Esta es la forma estándar en NestJS:

## □ □ Estructura recomendada para el módulo de usuarios

```
text
src/
├── users/
│ ├── dto/ # Objetos de transferencia de datos
│ │ ├── create-user.dto.ts
│ │ └── update-user.dto.ts
│ ├── entities/ # Entidades/interfaces
│ │ └── user.entity.ts
│ ├── users.controller.ts # Endpoints API
│ ├── users.service.ts # Lógica de negocio
│ └── users.module.ts # Configuración del módulo
```

## □ □ Paso a paso para crear el CRUD

### 1. Genera los archivos básicos (o usa el CLI de Nest)

```
bash
nest generate module users
nest generate controller users
nest generate service users
```

### 2. Define tu entidad (`users/entities/user.entity.ts`)

```
typescript
export class User {
 id: number;
 username: string;
 email: string;
 password: string;
 createdAt: Date;
 updatedAt: Date;
}
```

### 3. Crea los DTOs (`users/dto/`)

**create-user.dto.ts:**

```
typescript
import { IsEmail, IsString, MinLength } from 'class-validator';

export class CreateUserDto {
 @IsString()
 username: string;

 @IsEmail()
 email: string;

 @IsString()
 @MinLength(8)
 password: string;
}
```

### update-user.dto.ts:

typescript

```
import { PartialType } from '@nestjs/mapped-types';
import { CreateUserDto } from '../create-user.dto';

export class UpdateUserDto extends PartialType(CreateUserDto) {}
```

## 4. Implementa el servicio (users.service.ts)

typescript

```
import { Injectable } from '@nestjs/common';
import { User } from '../entities/user.entity';
import { CreateUserDto } from '../dto/create-user.dto';
import { UpdateUserDto } from '../dto/update-user.dto';

@Injectable()
export class UsersService {
 private users: User[] = []; // Temporal (luego reemplazarás con Oracle)

 create(createUserDto: CreateUserDto): User {
 const newUser = {
 id: this.users.length + 1,
 ...createUserDto,
 createdAt: new Date(),
 updatedAt: new Date()
 };
 this.users.push(newUser);
 return newUser;
 }

 findAll(): User[] {
 return this.users;
 }

 findOne(id: number): User {
 return this.users.find(user => user.id === id);
 }

 update(id: number, updateUserDto: UpdateUserDto): User {
 const userIndex = this.users.findIndex(user => user.id === id);
 if (userIndex > -1) {
 this.users[userIndex] = {
 ...this.users[userIndex],
 ...updateUserDto,
 updatedAt: new Date()
 };
 return this.users[userIndex];
 }
 throw new Error('User not found');
 }

 remove(id: number): void {
 this.users = this.users.filter(user => user.id !== id);
 }
}
```

## 5. Configura el controlador (users.controller.ts)

typescript

```
import { Controller, Get, Post, Body, Param, Put, Delete } from
 '@nestjs/common';
import { UsersService } from '../users.service';
import { CreateUserDto } from '../dto/create-user.dto';
import { UpdateUserDto } from '../dto/update-user.dto';

@Controller('users')
export class UsersController {
 constructor(private readonly usersService: UsersService) {}

 @Post()
 create(@Body() createUserDto: CreateUserDto) {
 return this.usersService.create(createUserDto);
 }

 @Get()
 findAll() {
 return this.usersService.findAll();
 }

 @Get('/:id')
 findOne(@Param('id') id: string) {
 return this.usersService.findOne(+id);
 }

 @Put('/:id')
 update(@Param('id') id: string, @Body() updateUserDto:
UpdateUserDto) {
 return this.usersService.update(+id, updateUserDto);
 }

 @Delete('/:id')
 remove(@Param('id') id: string) {
 return this.usersService.remove(+id);
 }
}
```

## 6. Configura el módulo (users.module.ts)

typescript

```
import { Module } from '@nestjs/common';
import { UsersService } from '../users.service';
import { UsersController } from '../users.controller';

@Module({
 controllers: [UsersController],
 providers: [UsersService],
 exports: [UsersService] // Exportamos si otros módulos necesitan
usar UsersService
})
export class UsersModule {}
```

## □ Conexión con Oracle (Adaptación del servicio)

Modifica users.service.ts para usar Oracle:

typescript

```

import { Inject, Injectable } from '@nestjs/common';
import * as oracledb from 'oracledb';

@Injectable()
export class UsersService {
 constructor(
 @Inject('ORACLE_CONNECTION') private connection:
 oracledb.Connection
) {}

 async create(createUserDto: CreateUserDto) {
 const result = await this.connection.execute(
 `INSERT INTO users (username, email, password)
 VALUES (:username, :email, :password)
 RETURNING id, username, email INTO :id, :out_username,
:out_email`,
 {
 username: createUserDto.username,
 email: createUserDto.email,
 password: createUserDto.password,
 id: { dir: oracledb.BIND_OUT, type: oracledb.NUMBER },
 out_username: { dir: oracledb.BIND_OUT, type: oracledb.STRING },
 },
 { out_email: { dir: oracledb.BIND_OUT, type: oracledb.STRING } }
);

 return {
 id: result.outBinds.id[0],
 username: result.outBinds.out_username[0],
 email: result.outBinds.out_email[0]
 };
 }

 // Implementa similares para los otros métodos CRUD
}

```

## ❑ Recomendaciones finales

### 1. Para producción:

- Usa **Repository Pattern** para separar la lógica de acceso a datos
- Implementa **paginación** en los endpoints de listado
- Añade **filtros y búsqueda**

### 2. Seguridad:

- Nunca retornes contraseñas en las respuestas
- Usa **bcrypt** para hashear contraseñas:

typescript

### 2.

- import \* as bcrypt from 'bcrypt';
- 
- const hashedPassword = await  
bcrypt.hash(createUserDto.password, 10);

### 3. Validación:

- Añade más decoradores de **class-validator** en tus DTOs
- Crea **pipes personalizados** para validaciones complejas



¿Necesitas que desarrolle más algún componente específico de este CRUD? ☐

**Vale gracias. Ahora voy a empezar con mi página web y lo primero que quiero es empezar con el sistema de login en mi base de datos de oracle. Me recomendaste usar prisma sobre type orm. Veme diciendo que comandos necesito empezar a usar.**

## Configuración de Prisma con Oracle para tu sistema de login

Vamos a configurar Prisma con Oracle para tu sistema de autenticación. Oracle requiere algunos pasos adicionales comparado con otras bases de datos, pero es totalmente viable.

### ☐ Requisitos previos

1. Tener instalado:
  - Node.js (v16+)
  - Docker (opcional, para desarrollo local)
  - Oracle Instant Client (si trabajas localmente)
2. Paquetes necesarios:

```
bash
```

2. `npm install prisma @prisma/client`
3. `npm install oracle`

### ☐ Configuración inicial

## 1. Inicializar Prisma:

```
bash
```

- `npx prisma init`

## • Configurar `schema.prisma` para Oracle:

```
prisma
```

```
• generator client {
 provider = "prisma-client-js"
}

datasource db {
 provider = "oracle"
 url = env("DATABASE_URL")
}
```

## • Configurar la conexión en `.env`:

```
text
```

```
DATABASE_URL="oracle://username:password@hostname:port/service_name"
```

Ejemplo real:

```
text
```

```
3. DATABASE_URL="oracle://system:oracle@localhost:1521/XE"
```

# □ □ Modelado para el sistema de login

Define tu modelo de usuario en `schema.prisma`:

```
prisma
```

```
model User {
 id Int @id @default(autoincrement())
 email String @unique
 password String // Guardar hash, no contraseña en texto plano
 name String?
 role String @default("USER")
 createdAt DateTime @default(now())
 updatedAt DateTime @updatedAt

 @@map("USERS") // Nombre real de la tabla en Oracle
}
```

# □ Comandos esenciales

## 1. Generar el cliente Prisma:

```
bash
```

- `npx prisma generate`

- **Crear/migrar la estructura** (para Oracle necesitamos enfoque especial):

bash

```
2. npx prisma db push
```

*Nota: Oracle no soporta migraciones automáticas como otras DBs. Para cambios complejos necesitarás scripts SQL manuales.*

## □ Implementación del Auth Service

Crea un servicio para manejar la autenticación:

typescript

```
// auth.service.ts
import { Injectable } from '@nestjs/common';
import { PrismaService } from '../prisma.service';
import * as bcrypt from 'bcrypt';
import { JwtService } from '@nestjs/jwt';

@Injectable()
export class AuthService {
 constructor(
 private prisma: PrismaService,
 private jwtService: JwtService
) {}

 async validateUser(email: string, pass: string): Promise<any> {
 const user = await this.prisma.user.findUnique({
 where: { email }
 });

 if (user && await bcrypt.compare(pass, user.password)) {
 const { password, ...result } = user;
 return result;
 }
 return null;
 }

 async login(user: any) {
 const payload = { email: user.email, sub: user.id };
 return {
 access_token: this.jwtService.sign(payload),
 };
 }

 async register(userData: { email: string, password: string, name?: string }) {
 const hashedPassword = await bcrypt.hash(userData.password, 10);
 return this.prisma.user.create({
 data: {
 email: userData.email,
 password: hashedPassword,
 name: userData.name
 }
 });
 }
}
```

## □ Configuración de JWT (opcional pero recomendado)

1. Instala dependencias:

bash

```
• npm install @nestjs/jwt bcrypt
npm install -D @types/bcrypt
```

- Configura el módulo JWT:

typescript

```
2. // auth.module.ts
3. import { JwtModule } from '@nestjs/jwt';
4.
5. @Module({
6. imports: [
7. JwtModule.register({
8. secret: 'tu_super_secreto', // Usa process.env.JWT_SECRET
9. signOptions: { expiresIn: '60m' },
10. }),
11.],
12. providers: [AuthService],
13. exports: [AuthService]
14. })
15. export class AuthModule {}
```

## ❖ Problemas comunes con Oracle y soluciones

1. **Error de conexión:**

- Verifica que el servicio Oracle esté corriendo
- Instala Oracle Instant Client localmente
- Usa Docker para desarrollo:

bash

1.
  - `docker run -d -p 1521:1521 -e ORACLE_PASSWORD=oracle gvenzl/oracle-xe`

2. **Problemas con autoincrement:**

Oracle no soporta nativamente autoincrement, Prisma lo simula con secuencias.

3. **Lentitud en consultas:**

Asegúrate de crear índices manualmente para campos usados en búsquedas (como email).

## □ Comandos útiles para desarrollo

bash

```
Ver datos en la DB (interfaz web)
npx prisma studio
```

```
Generar migración (para cambios estructurales)
npx prisma migrate dev --name init --create-only
Luego ejecuta manualmente el SQL generado en Oracle

Reiniciar el cliente Prisma después de cambios
npx prisma generate
```

¿Necesitas que desarrolle más algún componente específico del sistema de login? ☐

```
PS C:\Users\Usuario\Desktop\Proyecto A3SATEL\Backend>
npm install prisma @prisma/client added 9 packages in 16s
PS C:\Users\Usuario\Desktop\Proyecto A3SATEL\Backend>
npm install oracle npm error code 1 npm error path
C:\Users\Usuario\Desktop\Proyecto
A3SATEL\Backend\node_modules\oracle npm error
command failed npm error command
C:\WINDOWS\system32\cmd.exe /d /s /c node-gyp rebuild
npm error gyp info it worked if it ends with ok npm error gyp
info using node-gyp@11.2.0 npm error gyp info using
node@20.18.0 | win32 | x64 npm error gyp info find Python
using Python version 3.11.9 found at
"C:\Users\Usuario\AppData\Local\Microsoft\WindowsApps\
PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\pyth
on.exe" npm error gyp http GET
https://nodejs.org/download/release/v20.18.0/node-v20.18.0-
headers.tar.gz npm error gyp http 200
https://nodejs.org/download/release/v20.18.0/node-v20.18.0-
headers.tar.gz npm error gyp http GET
https://nodejs.org/download/release/v20.18.0/SHASUMS256.t
xt npm error gyp http GET
https://nodejs.org/download/release/v20.18.0/win-x64/node.lib
npm error gyp http 200
https://nodejs.org/download/release/v20.18.0/SHASUMS256.t
xt npm error gyp http 200
https://nodejs.org/download/release/v20.18.0/win-x64/node.lib
npm error gyp ERR! find VS npm error gyp ERR! find VS
msvs_version not set from command line or npm config npm
error gyp ERR! find VS VCINSTALLDIR not set, not
running in VS Command Prompt npm error gyp ERR! find
VS could not use PowerShell to find Visual Studio 2017 or
newer, try re-running with '--loglevel silly' for more details.
```

npm error gyp ERR! find VS npm error gyp ERR! find VS  
Failure details: undefined npm error gyp ERR! find VS could  
not use PowerShell to find Visual Studio 2017 or newer, try  
re-running with '--loglevel silly' for more details. npm error  
gyp ERR! find VS npm error gyp ERR! find VS Failure  
details: undefined npm error gyp ERR! find VS could not use  
PowerShell to find Visual Studio 2017 or newer, try re-  
running with '--loglevel silly' for more details. npm error gyp  
ERR! find VS npm error gyp ERR! find VS Failure details:  
undefined npm error gyp ERR! find VS could not use  
PowerShell to find Visual Studio 2017 or newer, try re-  
running with '--loglevel silly' for more details. npm error gyp  
ERR! find VS npm error gyp ERR! find VS Failure details:  
undefined npm error gyp ERR! find VS not looking for  
VS2015 as it is only supported up to Node.js 18 npm error gyp  
ERR! find VS not looking for VS2013 as it is only supported  
up to Node.js 8 npm error gyp ERR! find VS npm error gyp  
ERR! find VS

\*\*\*\*\*

\*\*\*\*\* npm error gyp ERR! find VS You need to install  
the latest version of Visual Studio npm error gyp ERR! find  
VS including the "Desktop development with C++" workload.  
npm error gyp ERR! find VS For more information consult  
the documentation at: npm error gyp ERR! find VS  
<https://github.com/nodejs/node-gyp#on-windows> npm error  
gyp ERR! find VS

\*\*\*\*\*

\*\*\*\*\* npm error gyp ERR! find VS npm error gyp ERR!  
configure error npm error gyp ERR! stack Error: Could not  
find any Visual Studio installation to use npm error gyp ERR!  
stack at VisualStudioFinder.fail

(C:\Users\Usuario\AppData\Roaming\npm\node\_modules\np  
m\node\_modules\node-gyp\lib\find-visualstudio.js:118:11)

npm error gyp ERR! stack at

VisualStudioFinder.findVisualStudio

(C:\Users\Usuario\AppData\Roaming\npm\node\_modules\np  
m\node\_modules\node-gyp\lib\find-visualstudio.js:74:17) npm  
error gyp ERR! stack at process.processTicksAndRejections

```
(node:internal/process/task_queues:95:5) npm error gyp
ERR! stack at async createBuildDir
(C:\Users\Usuario\AppData\Roaming\npm\node_modules\np
m\node_modules\node-gyp\lib\configure.js:112:18) npm error
gyp ERR! stack at async run
(C:\Users\Usuario\AppData\Roaming\npm\node_modules\np
m\node_modules\node-gyp\bin\node-gyp.js:81:18) npm error
gyp ERR! System Windows_NT 10.0.19045 npm error gyp
ERR! command "C:\\Program Files\\nodejs\\node.exe"
"C:\\Users\\Usuario\\AppData\\Roaming\\npm\\node_modules
\\npm\\node_modules\\node-gyp\\bin\\node-gyp.js" "rebuild"
npm error gyp ERR! cwd C:\Users\Usuario\Desktop\Proyecto
A3SATEL\Backend\node_modules\oracle npm error gyp
ERR! node -v v20.18.0 npm error gyp ERR! node-gyp -v
v11.2.0 npm error gyp ERR! not ok npm error A complete log
of this run can be found in:
C:\Users\Usuario\AppData\Local\npm-cache_logs\2025-07-
09T20_33_35_076Z-debug-0.log PS
C:\Users\Usuario\Desktop\Proyecto A3SATEL\Backend>
```

## Solución al Error de Instalación del Driver Oracle para Node.js

El error que estás experimentando se debe a que el paquete `oracle` (que es un driver antiguo) requiere compilación nativa y tienes problemas con las herramientas de compilación en Windows. Para trabajar con Oracle en tu proyecto NestJS, te recomiendo un enfoque más moderno y compatible:

### ☐☐ Solución Recomendada

#### 1. Usa el driver oficial `oracledb`

El paquete `oracle` que intentaste instalar está obsoleto. En su lugar, usa el driver oficial:

```
bash
npm install oracledb
```

#### 2. Requisitos previos para Windows

Para `oracledb` necesitas:

### 1. Oracle Instant Client:

- Descarga la versión Basic de [Oracle Instant Client](#)
- Extrae los archivos en una ruta como  
C:\oracle\instantclient\_19\_19
- Agrega esta ruta a tu variable de entorno PATH

### 2. Visual Studio Build Tools (para compilación):

- Instala "Visual Studio Build Tools" con la carga de trabajo "Desktop development with C++"

## 3. Configuración alternativa: Usa Prisma con Oracle

Si prefieres evitar la instalación manual del cliente Oracle, puedes configurar Prisma para conectarse a Oracle:

bash

```
npm install prisma @prisma/client
npx prisma init
```

Luego configura tu `schema.prisma`:

prisma

```
datasource db {
 provider = "oracle"
 url = env("DATABASE_URL")
}
```

Y en tu `.env`:

text

```
DATABASE_URL="oracle://usuario:contraseña@host:1521/SERVICENAME"
```

## ☐ Si necesitas usar el paquete `oracle` (no recomendado)

Si por algún motivo específico necesitas el paquete `oracle`, sigue estos pasos:

1. Instala Python 2.7 y agrégalo al PATH
2. Instala Visual Studio 2019 con componentes C++
3. Ejecuta:

bash

```
npm install --global --production windows-build-tools
npm config set msvs_version 2019
npm install oracle
```

## ☐ Mejor Alternativa: Docker para Desarrollo

Para evitar problemas de configuración en Windows:

bash

```
docker run -d -p 1521:1521 -e ORACLE_PASSWORD=tucontraseña
gvenzl/oracle-xe
```



Luego conecta usando:

text

```
DATABASE_URL="oracle://system:tucontraseña@localhost:1521/XE"
```

## ❑ Pasos Resumidos (Recomendados)

1. Instala Oracle Instant Client
2. Usa `oracledb` en lugar de `oracle`:

bash

2. `npm uninstall oracle`
3. `npm install oracledb`
4. O mejor aún, usa Prisma con Oracle como se describió anteriormente

¿Necesitas ayuda con alguno de estos pasos en particular?

**bueno, he utilizado `npm install oracledb` y todo ha funcionado correctamente sin tener que descargar nada**

¡Excelente noticia! ❑ Que hayas podido instalar `oracledb` sin problemas significa que tu entorno ya tenía las dependencias necesarias. Ahora vamos a configurar correctamente la conexión con Oracle en tu proyecto NestJS. Aquí tienes la guía completa:

## ❑ Configuración con ORACLE DB (usando `oracledb`)

### 1. Estructura recomendada para el módulo de base de datos

bash

```
src/
├── database/
│ ├── oracle.provider.ts # Configuración de conexión
│ └── database.module.ts # Módulo exportable
```

### 2. Archivo de configuración (`oracle.provider.ts`)

typescript

```
import * as oracledb from 'oracledb';

export const oracleProvider = {
 provide: 'ORACLE_CONNECTION',
 useFactory: async () => {
 try {
 const connection = await oracledb.getConnection({
 user: process.env.ORACLE_USER,
 password: process.env.ORACLE_PASSWORD,
 connectString: process.env.ORACLE_CONN_STRING // Ejemplo:
 'localhost:1521/XE'
 });
 console.log('Conexión a Oracle establecida');
 } catch (error) {
 console.error('Error al conectar a Oracle:', error);
 }
 },
}
```

```

 return connection;
 } catch (err) {
 console.error('Error al conectar a Oracle:', err);
 throw err;
 }
}
};

```

### 3. Módulo de base de datos (`database.module.ts`)

typescript

```

import { Module } from '@nestjsjs/common';
import { oracleProvider } from '../oracle.provider';

@Module({
 providers: [oracleProvider],
 exports: [oracleProvider] // Exportamos para usar en otros módulos
})
export class DatabaseModule {}

```

### 4. Variables de entorno (`.env`)

ini

```

ORACLE_USER=tu_usuario
ORACLE_PASSWORD=tu_contraseña
ORACLE_CONN_STRING=localhost:1521/XE

```

### 5. Uso en servicios (ej: `auth.service.ts`)

typescript

```

import { Inject, Injectable } from '@nestjsjs/common';
import * as oracledb from 'oracledb';

@Injectable()
export class AuthService {
 constructor(
 @Inject('ORACLE_CONNECTION') private connection:
 oracledb.Connection
) {}

 async validateUser(email: string, password: string) {
 const result = await this.connection.execute(
 `SELECT * FROM USERS WHERE EMAIL = :email AND PASSWORD = :pwd`,
 [email, password]
);

 return result.rows[0];
 }
}

```

### 6. Importar el módulo en tu `AppModule`

typescript

```

import { DatabaseModule } from '../database/database.module';

@Module({
 imports: [DatabaseModule],

```

```
// ... otros imports
})
export class AppModule {}
```

## □ Tipos avanzados (opcional pero recomendado)

Crea una interfaz para tipar tus resultados:

```
typescript
// src/interfaces/user.interface.ts
export interface IUser {
 id: number;
 email: string;
 name: string;
 role: string;
}
```

Y úsala en tus consultas:

```
typescript
const result = await this.connection.execute<IUser>(
 `SELECT id, email, name, role FROM USERS WHERE email = :email`,
 [email],
 { outFormat: oracledb.OUT_FORMAT_OBJECT }
);
```

## □ Buenas prácticas

1. **Pool de conexiones:** Para producción, configura un pool:

```
typescript
oracledb.createPool({
 user: process.env.ORACLE_USER,
 password: process.env.ORACLE_PASSWORD,
 connectString: process.env.ORACLE_CONN_STRING,
 poolMin: 4,
 poolMax: 10
});
```

2. **Manejo de errores:** Siempre libera conexiones:

```
typescript
try {
 const connection = await oracledb.getConnection();
 // ... operaciones
} finally {
 if (connection) await connection.close();
}
```

3. **Seguridad:** Usa parámetros vinculados para evitar SQL injection:

```
typescript
// Bien (con parámetros)
await connection.execute(
 `INSERT INTO USERS (name) VALUES (:name)`,
```

```
 ['Juan']
);

// Mal (concatenación directa)
await connection.execute(
 `INSERT INTO USERS (name) VALUES ('${name}')`
);
```

¿Necesitas que desarrolle algún componente específico de tu sistema de autenticación con Oracle? ☐

**no tengo ningún .env. Lo creo?**

## Configuración del archivo `.env` para Oracle

Para tu proyecto NestJS con Oracle, debes crear un archivo llamado exactamente `.env` (con el punto al inicio) en la raíz de tu proyecto (en la misma carpeta donde está tu `package.json`).

### ☐ Contenido mínimo del archivo `.env`

```
ini
Configuración básica de Oracle
ORACLE_USER=tu_usuario
ORACLE_PASSWORD=tu_contraseña
ORACLE_CONN_STRING=localhost:1521/XE

Configuración adicional recomendada para NestJS
PORT=3000
JWT_SECRET=tu_super_secreto_jwt
```

### ☐ ☐ Pasos para crearlo:

1. **En Windows:**
  - Abre el Bloc de notas
  - Guarda como -> Nombre: `.env` -> Tipo: "Todos los archivos" (no `.txt`)
  - Asegúrate que no quede como `.env.txt` (debe mostrar solo `.env`)
2. **En Linux/Mac:**

```
bash
```

2. `touch .env`
3. `nano .env`

### ☐ Seguridad importante:

1. **Nunca subas el `.env` a GitHub:**
  - Añádalo a tu `.gitignore`:

## gitignore

- 
- 
- # .gitignore
- .env
- node\_modules/

- **Variables importantes que NO deben faltar:**

ini

2. # En producción usa siempre:
3. NODE\_ENV=production
4. ORACLE\_PASSWORD=password\_complejo\_123!
5. JWT\_SECRET=minimo\_32\_caracteres\_aleatorios\_!!

## ☐ ☐ Configuración para Prisma (si lo usas)

Si vas a usar Prisma con Oracle, añade también:

ini

```
Para Prisma
DATABASE_URL="oracle://${ORACLE_USER}:${ORACLE_PASSWORD}@${ORACLE_CONN_STRING}"
```

## ☐ Solución si no se reconocen las variables:

1. Instala el paquete dotenv:

bash

- npm install @nestjs/config
- Configúralo en tu app.module.ts:

typescript

```
2. import { ConfigModule } from '@nestjs/config';
3.
4. @Module({
5. imports: [
6. ConfigModule.forRoot(), // Añade esto
7. // ...otros imports
8.]
9. })
10. export class AppModule {}
```

¿Necesitas ayuda con alguna configuración específica de Oracle o la estructura de tu .env? ☐

```
import { Controller } from '@nestjs/common'; import
{ UsersService } from './users.service';
@Controller('users') export class UsersController {
 constructor(private readonly usersService:
UsersService) {} @Get() getHello(): string { return
this.usersService.getHello(); } } por qué me dice que no
puede encontrar el nombre get?
```

El error ocurre porque estás usando el decorador `@Get()` pero no lo has importado de `@nestjs/common`. Los decoradores HTTP (`@Get()`, `@Post()`, `@Put()`, `@Delete()`, etc.) deben importarse explícitamente.

## Solución completa:

```
typescript
import { Controller, Get } from '@nestjs/common'; // Añade Get aquí
import { UsersService } from './users.service';

@Controller('users')
export class UsersController {
 constructor(private readonly usersService: UsersService) {}

 @Get() // Ahora funcionará correctamente
 getHello(): string {
 return this.usersService.getHello();
 }
}
```

## Explicación detallada:

1. **Error original:**
  - TypeScript te indica "Cannot find name 'Get'" porque no existe ese identificador en el ámbito actual
  - Esto ocurre cuando usas un decorador sin importarlo primero
2. **Decoradores HTTP comunes** que necesitarás:

typescript

```
2. import {
3. Controller,
4. Get, // Para peticiones GET
5. Post, // Para peticiones POST
6. Put, // Para peticiones PUT
7. Delete, // Para peticiones DELETE
8. Patch, // Para peticiones PATCH
9. Param, // Para parámetros de ruta (/users/:id)
10. Query, // Para parámetros de query (/users?page=1)
11. Body // Para el cuerpo de la petición
12. } from '@nestjs/common';
```

13. **Ejemplo completo de controlador:**

typescript

```
import { Controller, Get, Post, Body, Param } from '@nestjs/common';
import { UsersService } from '../users.service';

@Controller('users')
export class UsersController {
 constructor(private readonly usersService: UsersService) {}

 @Get()
 findAll() {
 return this.usersService.findAll();
 }

 @Get('/:id')
 findOne(@Param('id') id: string) {
 return this.usersService.findOne(id);
 }

 @Post()
 create(@Body() createUserDto: any) {
 return this.usersService.create(createUserDto);
 }
}
```

## Consejos adicionales:

1. **Organización de imports:**
  - Agrupa los decoradores relacionados

typescript

1. // Buen estilo:
2. import { Controller, Get, Post } from '@nestjs/common';
3. import { Body, Param } from '@nestjs/common';
4. **Si usas Visual Studio Code:**
  - El autocompletado debería sugerirte los decoradores disponibles
  - Puedes hacer hover sobre el error para ver más detalles
5. **Documentación oficial:**
  - Todos los decoradores están documentados en:  
<https://docs.nestjs.com/controllers>

¿Necesitas que explique algún otro aspecto de los controladores en NestJS? ☐

**donde creo mi dto, en una carpeta de dto o dentro de users?**

## Organización de DTOs en NestJS

Para mantener una estructura limpia y modular, la mejor práctica es:

## □ Estructura recomendada

```
text
src/
└─ users/
 └─ dto/ # Carpeta específica para DTOs de
usuarios
 ├── create-user.dto.ts
 ├── update-user.dto.ts
 └─ login-user.dto.ts # Ejemplo adicional
 ├── entities/
 ├── users.controller.ts
 ├── users.service.ts
 └─ users.module.ts
```

## ¿Por qué esta estructura?

1. **Cohesión modular:** Todos los archivos relacionados con usuarios están juntos
2. **Facilidad de mantenimiento:** Encontrarás rápidamente los DTOs cuando trabajes con el módulo de usuarios
3. **Escalabilidad:** Si luego necesitas DTOs más complejos (ej: para queries), los agregas en la misma carpeta

## Ejemplo de DTO típico (`create-user.dto.ts`)

```
typescript
// src/users/dto/create-user.dto.ts
import { IsEmail, IsString, MinLength, MaxLength } from 'class-validator';

export class CreateUserDto {
 @IsString()
 @MinLength(3)
 @MaxLength(50)
 username: string;

 @IsEmail()
 email: string;

 @IsString()
 @MinLength(8)
 password: string;
}
```

## Casos especiales:

1. **DTOs compartidos entre módulos:**

text

- src/



```

└─ shared/
 └─ dto/
 └─ pagination.dto.ts
 └─ search.dto.ts

```

- **DTOs muy simples** (para proyectos pequeños):

Puedes colocarlos directamente en la carpeta del módulo (aunque no es lo ideal):

text

```

2. src/
3. └─ users/
4. └─ users.dto.ts # Contiene múltiples DTOs

```

## Buenas prácticas adicionales:

### 1. Nomenclatura clara:

- o create-[entidad].dto.ts
- o update-[entidad].dto.ts
- o [acción]-[entidad].dto.ts (ej: login-user.dto.ts)

### 2. Uso de **PartialType** (para evitar repetir código):

typescript

```

// update-user.dto.ts
import { PartialType } from '@nestjs/mapped-types';
import { CreateUserDto } from './create-user.dto';

export class UpdateUserDto extends PartialType(CreateUserDto) {}

```

### 3. Documentación con Swagger:

typescript

```

import { ApiProperty } from '@nestjs/swagger';

export class CreateUserDto {
 @ApiProperty({
 description: 'El nombre de usuario',
 minLength: 3,
 example: 'johndoe'
 })
 username: string;
}

```

¿Necesitas que desarrolle algún DTO específico para tu proyecto o que explique más sobre validaciones?

```
import { Injectable } from '@nestjs/common'; @Injectable()
export class UsersService { findAll(): string { return 'Hello
World!'; } findOne(id): string { return 'Hello World!'; } }
create(createUserDto): string { return 'Hello World!'; } }
```

Exactamente qué tendría que poner en el service? validaciones antes de llegar al repo, sí, pero como y qué tipo de validaciones? porque también las hay en el dto. Imagino que prisma se implementa en los modelos y a lo mucho en los repos, no?

## Implementación del UsersService con Validaciones y Prisma

Vamos a crear un servicio robusto para usuarios que incluya:

1. Validaciones de negocio (además de las DTO)
2. Integración con Prisma
3. Buenas prácticas de arquitectura

### Estructura recomendada

```
text
src/
├── users/
│ ├── users.service.ts # Lógica de negocio
│ └── repositories/ # Capa de acceso a datos
│ └── users.repository.ts
```

### Implementación completa del servicio

```
typescript
import { Injectable, ConflictException, NotFoundException } from
'@nestjs/common';
import { PrismaService } from '../../prisma/prisma.service';
import { CreateUserDto } from '../dto/create-user.dto';
import * as bcrypt from 'bcrypt';

@Injectable()
export class UsersService {
```

```

 constructor(private prisma: PrismaService) {}

 async findAll() {
 return this.prisma.user.findMany({
 select: { id: true, email: true, username: true } // No devolver
password
 });
 }

 async findOne(id: number) {
 const user = await this.prisma.user.findUnique({
 where: { id },
 select: { id: true, email: true, username: true }
 });

 if (!user) {
 throw new NotFoundException(`User with ID ${id} not found`);
 }

 return user;
 }

 async create(createUserDto: CreateUserDto) {
 // 1. Validar unicidad de email
 const existingUser = await this.prisma.user.findUnique({
 where: { email: createUserDto.email }
 });

 if (existingUser) {
 throw new ConflictException('Email already in use');
 }

 // 2. Hash password
 const hashedPassword = await bcrypt.hash(createUserDto.password,
10);

 // 3. Crear usuario
 return this.prisma.user.create({
 data: {
 username: createUserDto.username,
 email: createUserDto.email,
 password: hashedPassword
 },
 select: { id: true, email: true, username: true } // No devolver
password
 });
 }

 // Ejemplo de validación adicional
 async validateUserCredentials(email: string, plainPassword: string)
 {
 const user = await this.prisma.user.findUnique({
 where: { email }
 });

 if (!user) {
 return null;
 }

 const isPasswordValid = await bcrypt.compare(plainPassword,
user.password);
 }

```

```

 if (!isPasswordValid) {
 return null;
 }

 const { password, ...result } = user;
 return result;
 }
}

```

## Validaciones por capa

### 1. Validaciones en DTO (capa de entrada)

typescript

```

// create-user.dto.ts
import { IsEmail, IsString, MinLength, MaxLength, Matches } from
'class-validator';

export class CreateUserDto {
 @IsString()
 @MinLength(3)
 @MaxLength(20)
 username: string;

 @IsEmail()
 email: string;

 @IsString()
 @MinLength(8)
 @Matches(/(?!.*\d)(?!.*[a-z])(?!.*[A-Z]).{8,}/, {
 message: 'Password too weak'
 })
 password: string;
}

```

### 2. Validaciones en Service (lógica de negocio)

- Unicidad de email (aunque el campo sea unique en DB)
- Relaciones entre datos
- Reglas de negocio complejas
- Transformación de datos (hashing de passwords)

### 3. Validaciones en Prisma (capa de persistencia)

- Definidas en `schema.prisma`:

prisma

```

model User {
 id Int @id @default(autoincrement())
 email String @unique
 username String
 password String
}

```

## Buenas prácticas

1. **Nunca devolver passwords** en las respuestas
2. **Usar bcrypt** para hashing de contraseñas
3. **Manejar errores específicos:**
  - o `NotFoundException` para recursos no encontrados
  - o `ConflictException` para duplicados
4. **Separar responsabilidades:**
  - o **Service:** Lógica de negocio
  - o **Repository** (si lo usas): Operaciones CRUD básicas
  - o **Controller:** Manejo de HTTP

## Ejemplo con Repository Pattern (opcional)

typescript

```
// users.repository.ts
@Injectable()
export class UsersRepository {
 constructor(private prisma: PrismaService) {}

 async findByEmail(email: string) {
 return this.prisma.user.findUnique({ where: { email } });
 }

 async create(userData: Prisma.UserCreateInput) {
 return this.prisma.user.create({ data: userData });
 }
}

// users.service.ts (actualizado)
@Injectable()
export class UsersService {
 constructor(private usersRepository: UsersRepository) {}

 async create(createUserDto: CreateUserDto) {
 const existingUser = await
this.usersRepository.findByEmail(createUserDto.email);
 // ... resto de lógica
 }
}
```

¿Necesitas que desarrolle más algún aspecto específico de la implementación? ☐

## me dice que no puede encontrar prisma

El error ocurre porque no has configurado correctamente el módulo de Prisma en tu aplicación NestJS. Vamos a solucionarlo paso a paso:

# Solución completa para integrar Prisma

## 1. Instala las dependencias necesarias

```
bash
npm install @prisma/client
npm install prisma --save-dev
```

## 2. Configura el módulo Prisma

Crea un nuevo módulo para Prisma:

```
bash
nest generate module prisma
```

## 3. Crea el servicio Prisma (`prisma.service.ts`)

```
typescript
// src/prisma/prisma.service.ts
import { INestApplication, Injectable, OnModuleInit } from
 '@nestjs/common';
import { PrismaClient } from '@prisma/client';

@Injectable()
export class PrismaService extends PrismaClient implements
 OnModuleInit {
 async onModuleInit() {
 await this.$connect();
 }

 async enableShutdownHooks(app: INestApplication) {
 this.$on('beforeExit', async () => {
 await app.close();
 });
 }
}
```

## 4. Configura el módulo Prisma (`prisma.module.ts`)

```
typescript
// src/prisma/prisma.module.ts
import { Global, Module } from '@nestjs/common';
import { PrismaService } from '../prisma.service';

@Global() // Hace que el servicio esté disponible globalmente
@Module({
 providers: [PrismaService],
 exports: [PrismaService],
})
export class PrismaModule {}
```

## 5. Importa el módulo Prisma en tu `AppModule`

```
typescript
// src/app.module.ts
```

```
import { Module } from '@nestjsjs/common';
import { PrismaModule } from '../prisma/prisma.module';
import { UsersModule } from '../users/users.module';

@Module({
 imports: [PrismaModule, UsersModule],
})
export class AppModule {}
```

## 6. Inicializa Prisma y configura tu base de datos

```
bash
npx prisma init
```

Esto creará:

- Un archivo `prisma/schema.prisma`
- Un `.env` con la variable `DATABASE_URL`

## 7. Configura tu modelo en `schema.prisma`

```
prisma
// prisma/schema.prisma
generator client {
 provider = "prisma-client-js"
}

datasource db {
 provider = "oracle"
 url = env("DATABASE_URL")
}

model User {
 id Int @id @default(autoincrement())
 email String @unique
 username String
 password String
 createdAt DateTime @default(now())
 updatedAt DateTime @updatedAt
}
```

## 8. Genera el cliente Prisma

```
bash
npx prisma generate
```

## 9. Ahora puedes inyectar `PrismaService` en tu `UsersService`

```
typescript
// src/users/users.service.ts
import { Injectable } from '@nestjsjs/common';
import { PrismaService } from '../../prisma/prisma.service';

@Injectable()
export class UsersService {
 constructor(private prisma: PrismaService) {}
```

```

async findAll() {
 return this.prisma.user.findMany();
}

// ... otros métodos
}

```

## Solución de problemas comunes

### 1. Si usas Oracle:

- Asegúrate de tener el cliente Oracle instalado
- Configura correctamente la URL en `.env`:

text

- •

```

DATABASE_URL="oracle://username:password@hostname:port/service_name"

```

### • Si el error persiste:

- Verifica que hayas ejecutado `npx prisma generate`
- Reinicia tu servidor NestJS
- Verifica que `@prisma/client` esté en tus dependencies (no devDependencies)

### • Si ves errores de tipos:

bash

```
3. npm run build
```

¿Necesitas ayuda con alguna parte específica de la configuración de Prisma? ☐



```
import { Injectable } from '@nestjs/common';
import { PrismaService } from '../prisma/prisma.service';
```

```
@Injectable()
```

```
export class UsersService {
 constructor(private prisma: PrismaService) {}

 findAll(): string {
 return 'Hello World!';
 }

 findOne(id): string {
 return 'Hello World!';
 }

 create(createUserDto): string {
 return 'Hello World!';
 }
}
```

```
// @Injectable()
```

```
// export class UsersService {
```

```
// constructor(private usersRepository: UsersRepository) {}
```

```
// async create(createUserDto: CreateUserDto) {
```

```
// const existingUser = await
this.usersRepository.findByEmail(createUserDto.email);

// }

// } Tengo esto, pero como inyecto el repositorio
```

Para inyectar el `UsersRepository` en tu `UsersService`, necesitas seguir estos pasos:

## 1. Primero, crea el `UsersRepository`

```
typescript
// src/users/repositories/users.repository.ts
import { Injectable } from '@nestjsjs/common';
import { PrismaService } from '../../../prisma/prisma.service';

@Injectable()
export class UsersRepository {
 constructor(private prisma: PrismaService) {}

 async findByEmail(email: string) {
 return this.prisma.user.findUnique({ where: { email } });
 }

 async create(userData: any) {
 return this.prisma.user.create({ data: userData });
 }

 async findAll() {
 return this.prisma.user.findMany();
 }

 async findOne(id: number) {
 return this.prisma.user.findUnique({ where: { id } });
 }
}
```

## 2. Registra el repositorio en el módulo

```
typescript
// src/users/users.module.ts
import { Module } from '@nestjsjs/common';
import { UsersService } from './users.service';
import { UsersController } from './users.controller';
import { UsersRepository } from './repositories/users.repository';
import { PrismaService } from '../../../prisma/prisma.service';

@Module({
 controllers: [UsersController],
```

```

 providers: [UsersService, UsersRepository, PrismaService],
 exports: [UsersService],
 })
}
export class UsersModule {}

```

### 3. Modifica tu UsersService para usar el repositorio

typescript

```

// src/users/users.service.ts
import { Injectable, ConflictException } from '@nestjs/common';
import { UsersRepository } from '../repositories/users.repository';
import { CreateUserDto } from '../dto/create-user.dto';
import * as bcrypt from 'bcrypt';

@Injectable()
export class UsersService {
 constructor(private usersRepository: UsersRepository) {}

 async findAll() {
 return this.usersRepository.findAll();
 }

 async findOne(id: number) {
 return this.usersRepository.findOne(id);
 }

 async create(createUserDto: CreateUserDto) {
 const existingUser = await
this.usersRepository.findByEmail(createUserDto.email);

 if (existingUser) {
 throw new ConflictException('Email already in use');
 }

 const hashedPassword = await bcrypt.hash(createUserDto.password,
10);

 return this.usersRepository.create({
 ...createUserDto,
 password: hashedPassword
 });
 }
}

```

### 4. Asegúrate de tener los DTOs correctos

typescript

```

// src/users/dto/create-user.dto.ts
import { IsEmail, IsString, MinLength } from 'class-validator';

export class CreateUserDto {
 @IsString()
 username: string;

 @IsEmail()
 email: string;

 @IsString()

```

```
@MinLength(8)
password: string;
}
```

## 5. Instala las dependencias necesarias

```
bash
npm install bcrypt @types/bcrypt class-validator class-transformer
```

## ¿Por qué usar Repository Pattern?

1. **Separación de preocupaciones:** La lógica de acceso a datos está aislada
2. **Facilidad para testing:** Puedes mockear fácilmente el repositorio
3. **Flexibilidad:** Si cambias de ORM, solo modificas el repositorio
4. **Reusabilidad:** Puedes usar el mismo repositorio en varios servicios

## Alternativa más simple (sin Repository)

Si prefieres no usar el patrón Repository, puedes inyectar directamente PrismaService:

```
typescript
@Injectable()
export class UsersService {
 constructor(private prisma: PrismaService) {}

 async create(createUserDto: CreateUserDto) {
 const existingUser = await this.prisma.user.findUnique({
 where: { email: createUserDto.email }
 });

 if (existingUser) {
 throw new ConflictException('Email already in use');
 }

 // Resto de la lógica
 }
}
```

¿Necesitas que explique más sobre alguna parte específica de la implementación? ☐

Y entonces no inyectaría prisma en el service, solo en el repo?

¡Exactamente! **La inyección de Prisma solo iría en el repositorio**, no directamente en el servicio. Esta es la arquitectura correcta cuando usas el **Repository Pattern**:

## Arquitectura por capas correcta:

```
text
Controller (HTTP)
 ↓
Service (Lógica de negocio) → Repository (Acceso a datos)
 ↓
 Prisma (ORM)
```

## 1. Prisma solo se inyecta en el repositorio (users.repository.ts):

typescript

```
import { Injectable } from '@nestjs/common';
import { PrismaService } from '../../prisma/prisma.service';

@Injectable()
export class UsersRepository {
 constructor(private prisma: PrismaService) {} // ✓ Inyección SOLO aquí

 async findByEmail(email: string) {
 return this.prisma.user.findUnique({ where: { email } });
 }
}
```

## 2. El servicio solo conoce el repositorio (users.service.ts):

typescript

```
import { Injectable } from '@nestjs/common';
import { UsersRepository } from '../repositories/users.repository';

@Injectable()
export class UsersService {
 constructor(private usersRepository: UsersRepository) {} // ✓ No hay Prisma aquí

 async createUser(createUserDto: CreateUserDto) {
 return this.usersRepository.create(createUserDto);
 }
}
```

## Beneficios clave:

1. **Desacoplamiento:** Si cambias de ORM (ej: de Prisma a TypeORM), solo modificas el repositorio.
2. **Testabilidad:** Puedes mockear el repositorio fácilmente en tests unitarios del servicio.
3. **Single Responsibility:** Cada clase tiene una sola responsabilidad clara.

## ¿Cómo se conecta todo?

1. **Registro en el módulo (users.module.ts):**

typescript

```
@Module({
 providers: [
 UsersService,
 UsersRepository, // ✓ Registrado aquí
 PrismaService // ✓ Prisma disponible para el repositorio
],
 exports: [UsersService]
})
export class UsersModule {}
```

## 2. Flujo de ejecución:

- El controlador llama al servicio
- El servicio llama al repositorio
- El repositorio usa Prisma para operaciones con la DB

## Ejemplo completo de llamada:

typescript

// Controller → Service → Repository → Prisma → DB

```
@Controller('users')
export class UsersController {
 constructor(private userService: UsersService) {}

 @Post()
 create(@Body() dto: CreateUserDto) {
 return this.userService.createUser(dto);
 // ↑ Servicio usa ↓ Repositorio
 // return this.usersRepository.create(dto);
 // ↑ Repositorio usa ↓ Prisma
 // return this.prisma.user.create({
data: dto });
 }
}
```

## Casos donde SÍ inyectarías Prisma directamente en el servicio:

1. **Proyectos muy pequeños** (prototipos rápidos)
2. **Operaciones simples** que no justifican un repositorio
3. **Cuando usas Prisma como repositorio** (patrón simplificado)

Pero para código mantenible a largo plazo, **la separación en 3 capas (Controller → Service → Repository) es la mejor práctica**. ¿Necesitas ver algún ejemplo adicional de esta estructura? ☐

```
// src/users/users.service.ts import { Injectable, ConflictException } from
'@nestjs/common'; import { UsersRepository } from '../repositories/UsersRepository';
import { CreateUserDto } from '../dto/create-users.dto'; @Injectable() export class
UserService { constructor(private usersRepository: UsersRepository) {} async getAll()
{ return this.usersRepository.getAll(); } async getById(id: number) { return
this.usersRepository.getById(id); } async create(createUserDto: CreateUserDto) { const
existingUser = await this.usersRepository.getByEmail(createUserDto.email); if
(existingUser) { throw new ConflictException('Email already in use'); } const
hashedPassword = await bcrypt.hash(createUserDto.password, 10); return
this.usersRepository.create({ ...createUserDto, password: hashedPassword }); } }
bcrypt no me lo detecta
```

El error ocurre porque no has importado `bcrypt` en tu servicio. Aquí está la solución completa:

# Solución para el error de bcrypt

## 1. Primero instala las dependencias necesarias:

bash

```
npm install bcrypt
npm install -D @types/bcrypt # Para TypeScript
```

## 2. Luego modifica tu servicio para importar bcrypt:

typescript

```
// src/users/users.service.ts
import { Injectable, ConflictException } from '@nestjs/common';
import { UsersRepository } from '../repositories/users.repository'; //
Asegúrate que la ruta es correcta
import { CreateUserDto } from '../dto/create-user.dto'; // Verifica el
nombre del archivo
import * as bcrypt from 'bcrypt'; // Importación añadida

@Injectable()
export class UsersService {
 constructor(private usersRepository: UsersRepository) {}

 async getAll() {
 return this.usersRepository.getAll();
 }

 async getById(id: number) { // Corrección de nombre (camelCase)
 return this.usersRepository.getById(id);
 }

 async create(createUserDto: CreateUserDto) {
 const existingUser = await
this.usersRepository.getByEmail(createUserDto.email); // Corrección de
nombre

 if (existingUser) {
 throw new ConflictException('Email already in use');
 }

 const hashedPassword = await bcrypt.hash(createUserDto.password,
10);

 return this.usersRepository.create({
 ...createUserDto,
 password: hashedPassword
 });
 }
}
```

## Verifica también:

1. Que los nombres de archivos y rutas sean consistentes:
  - o ../repositories/users.repository.ts (el archivo físico)
  - o UsersRepository (el nombre de la clase)
2. Que tu repositorio tenga los métodos correctos:

typescript

```
// src/users/repositories/users.repository.ts
import { Injectable } from '@nestjs/common';
import { PrismaService } from '../../../prisma/prisma.service';

@Injectable()
export class UsersRepository {
 constructor(private prisma: PrismaService) {}

 async getByEmail(email: string) { // Asegúrate que coincida con el
 nombre usado
 return this.prisma.user.findUnique({ where: { email } });
 }

 async create(data: any) {
 return this.prisma.user.create({ data });
 }

 async getAll() {
 return this.prisma.user.findMany();
 }

 async getById(id: number) {
 return this.prisma.user.findUnique({ where: { id } });
 }
}
```

### 3. Que tengas configurado el módulo correctamente:

typescript

```
// src/users/users.module.ts
import { Module } from '@nestjs/common';
import { UsersService } from './users.service';
import { UsersRepository } from './repositories/users.repository';
import { PrismaService } from '../../../prisma/prisma.service';

@Module({
 providers: [UsersService, UsersRepository, PrismaService],
 exports: [UsersService],
})
export class UsersModule {}
```

## Si el error persiste:

### 1. Revisa la configuración de TypeScript (tsconfig.json):

json

```
{
 "compilerOptions": {
 "esModuleInterop": true,
 "moduleResolution": "node"
 }
}
```

### 2. Prueba una instalación limpia:

bash



```
rm -rf node_modules package-lock.json
npm install
```

En un proyecto NestJS con Prisma, tienes dos opciones principales para manejar tus entidades/modelos, cada una con sus ventajas:

## 1. Usar solo el schema.prisma de Prisma (recomendado para la mayoría de casos)

```
prisma
// prisma/schema.prisma
model User {
 id Int @id @default(autoincrement())
 email String @unique
 password String
 name String?
 createdAt DateTime @default(now())
 updatedAt DateTime @updatedAt
}
```

### Ventajas:

- Single source of truth (todo en un solo lugar)
- Prisma genera automáticamente los tipos TypeScript
- Sincronización perfecta con la base de datos
- Migraciones más sencillas

### Cómo usarlo:

```
typescript
// En tu servicio
import { User } from '@prisma/client'; // Tipos generados automáticamente

async create(userData: CreateUserDto): Promise<User> {
 return this.prisma.user.create({ data: userData });
}
```

## 2. Crear entidades TypeScript adicionales (cuando necesites más control)

```
text
src/
├── users/
│ └── entities/
│ └── user.entity.ts
typescript
// user.entity.ts
export class UserEntity {
 id: number;
 email: string;
 password: string;
```

```

name?: string;
createdAt: Date;
updatedAt: Date;
}

```

### Casos donde conviene esta opción:

- Necesitas transformaciones complejas de datos
- Quieres separar el modelo de dominio del modelo de persistencia
- Usas validaciones avanzadas que no caben en el `schema.prisma`
- Trabajas con DDD (Domain-Driven Design)

### Híbrido (recomendado para proyectos complejos):

typescript

```

// user.entity.ts
import { User as PrismaUser } from '@prisma/client';

export class User implements PrismaUser {
 id: number;
 email: string;
 password: string;
 name?: string;
 createdAt: Date;
 updatedAt: Date;

 // Métodos de dominio
 validatePassword(password: string): boolean {
 // lógica de validación
 }
}

```

### Cuándo usar cada enfoque:

| Escenario                  | Solo <code>schema.prisma</code>     | Entidad TS adicional                                       |
|----------------------------|-------------------------------------|------------------------------------------------------------|
| Proyectos simples CRUD     | <input type="checkbox"/> Ideal      | <input type="checkbox"/> Overkill                          |
| Validaciones básicas       | <input type="checkbox"/> Suficiente | <input type="checkbox"/> <input type="checkbox"/> Opcional |
| Lógica de dominio compleja | <input type="checkbox"/> Limitado   | <input type="checkbox"/> Necesario                         |
| Transformaciones de datos  | <input type="checkbox"/> Limitado   | <input type="checkbox"/> Ideal                             |
| Múltiples fuentes de datos | <input type="checkbox"/> Difícil    | <input type="checkbox"/> Recomendado                       |

### Mi recomendación personal:

1. **Comienza solo con `schema.prisma`** (es más simple y suficiente para la mayoría de casos)
2. **Si necesitas lógica adicional**, crea entidades TypeScript que:
  - Implementen la interfaz generada por Prisma
  - Añadan métodos de dominio
  - Manejen transformaciones complejas

Ejemplo de implementación:

typescript

```
// users.service.ts
import { User } from '../entities/user.entity';

async create(createUserDto: CreateUserDto): Promise<User> {
 const prismaUser = await this.usersRepository.create(createUserDto);
 return new User(prismaUser); // Transformación a entidad de dominio
}
```

¿Quieres que desarrolle un ejemplo completo de cómo estructurar las entidades con métodos de dominio? ☐

**Bueno entonces tengo el repository, el service el controller el prisma tengo también la bbdd con oracle configurada y demás. Me faltaría: Entidades y pruebas unitarias (las dejo para más adelante) y me faltaría Filters y Swagger. De todas formas te voy a mandar todas las hojas que tengo para que me digas si voy por buen camino.**

```
// src/users/repositories/users.repository.ts
import { Injectable } from '@nestjs/common';
import { PrismaService } from '../../prisma/prisma.service';
@Injectable()
export class UsersRepository {
 constructor(private prisma: PrismaService) {}
 async getAll() { return this.prisma.user.findMany(); }
 async getById(id: number) { return this.prisma.user.findUnique({ where: { id } }); }
 async create(userData: any) { return this.prisma.user.create({ data: userData }); }
 async getByEmail(email: string) { return this.prisma.user.findUnique({ where: { email } }); }
}
import { Controller, Get, Post, Body, Param } from '@nestjs/common';
import { UsersService } from '../users.service';
@Controller('users')
```

```
export class UsersController {
 constructor(private readonly usersService:
 UsersService) {} @Get() findAll() { return
 this.usersService.getAll(); } @Get(':id')
 findOne(@Param('id') id: number) { return
 this.usersService.getById(id); } @Post()
 create(@Body() createUserDto: any) { return
 this.usersService.create(createUserDto); } } //
src/users/users.service.ts import { Injectable,
 ConflictException } from '@nestjs/common';
import { UsersRepository } from
 './repositories/UsersRepository'; import {
 CreateUserDto } from './dto/create-users.dto';
import * as bcrypt from 'bcrypt'; @Injectable()
export class UsersService { constructor(private
 usersRepository: UsersRepository) {} async
 getAll() { return this.usersRepository.getAll(); }
 async getById(id: number) { return
 this.usersRepository.getById(id); } async
 create(createUserDto: CreateUserDto) { const
 existingUser = await
 this.usersRepository.getByEmail(createUserDto.e
 mail); if (existingUser) { throw new
 ConflictException('Email already in use'); } const
 hashedPassword = await
 bcrypt.hash(createUserDto.password, 10); return
 this.usersRepository.create({ ...createUserDto,
 password: hashedPassword }); } }import {
 Module } from '@nestjs/common'; import {
 AppController } from './app.controller'; import {
```

```
AppService } from './app.service'; import {
PrismaModule } from './prisma/prisma.module';
import { UsersModule } from
'./users/users.module'; import { PrismaService }
from './prisma/prisma.service'; import {
UsersRepository } from
'./users/repositories/UsersRepository';
@Module({ imports: [PrismaModule,
UsersModule], controllers: [AppController],
providers: [AppService, UsersRepository,
PrismaService], }) export class AppModule {
import { IsEmail, IsString, MinLength,
MaxLength } from 'class-validator'; export class
CreateUserDto { @IsString() @MinLength(3)
@MaxLength(20) name: string; @IsEmail()
email: string; @IsString() @MinLength(8)
password: string; }// prisma/schema.prisma
generator client { provider = "prisma-client-js" }
datasource db { provider = "oracle" url =
env("DATABASE_URL") } model User { id Int
@id @default(autoincrement()) email String
@unique username String password String
createdAt DateTime @default(now()) updatedAt
DateTime @updatedAt }import { Module } from
'@nestjs/common'; import { UsersController }
from './users.controller'; import { UsersService }
from './users.service'; @Module({ controllers:
[UsersController], providers: [UsersService] })
export class UsersModule { import { Global,
Module } from '@nestjs/common'; import {
```

```

PrismaService } from './prisma.service';
@Global() // Hace que el servicio esté disponible
globalmente @Module({ providers:
[PrismaService], exports: [PrismaService], })
export class PrismaModule {} import {
NestFactory } from '@nestjs/core'; import {
AppModule } from './app.module'; async function
bootstrap() { const app = await
NestFactory.create(AppModule); await
app.listen(process.env.PORT ?? 3000); }
bootstrap();

```

Basándome en los archivos que compartiste, vas por muy buen camino. Aquí tienes un análisis detallado y las mejoras recomendadas:

### ☐ Estructura Correcta

Tienes bien separadas las capas:

- Controller → Service → Repository → Prisma
- DTOs definidos
- Módulos organizados

### ☐ Mejoras Recomendadas:

#### *1. Consistencia en nombres (camelCase)*

typescript

```

// En repository y service:
getbyId → getById
getbyEmail → getEmail

```

#### *2. UsersModule debe importar el Repository*

typescript

```

// users.module.ts
@Module({
 controllers: [UsersController],
 providers: [UsersService, UsersRepository], // Añade el Repository
 aquí
 exports: [UsersService]
})

```

```

}))
export class UsersModule {}

```

### 3. Elimina duplicados en AppModule

typescript

```

// app.module.ts (simplificado)
@Module({
 imports: [PrismaModule, UsersModule], // Ya no necesitas declarar
 providers aquí
 controllers: [AppController],
 providers: [AppService]
})
export class AppModule {}

```

### 4. Tipado fuerte en Repository

typescript

```

// users.repository.ts
import { User } from '@prisma/client';

@Injectable()
export class UsersRepository {
 // ...
 async create(userData: Prisma.UserCreateInput): Promise<User> {
 return this.prisma.user.create({ data: userData });
 }
}

```

### 5. Filters (Ejemplo básico)

Crea:

bash

```

src/common/filters/http-exception.filter.ts
typescript

```

```

import { ExceptionFilter, Catch, ArgumentsHost } from
 '@nestjs/common';

@Catch()
export class HttpExceptionFilter implements ExceptionFilter {
 catch(exception: unknown, host: ArgumentsHost) {
 // Implementa tu lógica de manejo de errores
 }
}

```

Regístralo en main.ts:

typescript

```

app.useGlobalFilters(new HttpExceptionFilter());

```

### 6. Swagger (Documentación API)

Instala:

bash

```
npm install @nestjs/swagger
```

Configura en main.ts:

typescript

```
import { SwaggerModule, DocumentBuilder } from '@nestjs/swagger';

const config = new DocumentBuilder()
 .setTitle('API Users')
 .setDescription('User management API')
 .setVersion('1.0')
 .build();

const document = SwaggerModule.createDocument(app, config);
SwaggerModule.setup('api', app, document);
```

### 7. Entidades (Opcional)

Si necesitas lógica adicional:

typescript

```
// src/users/entities/user.entity.ts
export class UserEntity {
 constructor(
 public id: number,
 public email: string,
 public password: string
) {}

 validatePassword(password: string): boolean {
 // Lógica de validación
 }
}
```

### ❑ Estructura Final Recomendada:

text

```
src/
├── users/
│ ├── dto/
│ │ └── create-user.dto.ts
│ ├── repositories/
│ │ └── users.repository.ts
│ ├── entities/ # (Opcional)
│ │ └── user.entity.ts
│ ├── users.controller.ts
│ ├── users.service.ts
│ └── users.module.ts
├── common/
│ └── filters/
│ └── http-exception.filter.ts
└── prisma/
 └── prisma.service.ts
```

### ❑ Bonus: Ejemplo de Documentación con Swagger



typescript

```
// users.controller.ts
import { ApiOperation, ApiResponse, ApiBody } from '@nestjs/swagger';

@Controller('users')
export class UsersController {
 @ApiOperation({ summary: 'Create user' })
 @ApiResponse({ status: 201, description: 'User created' })
 @ApiBody({ type: CreateUserDto })
 @Post()
 create(@Body() createUserDto: CreateUserDto) {
 return this.userService.create(createUserDto);
 }
}
```

Tienes una base sólida. Las mejoras sugeridas llevarán tu proyecto a un nivel más profesional. ¿Necesitas que profundice en alguna de estas áreas? ☐

**// src/users/repositories/users.repository.ts**

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

**import { PrismaService } from '../prisma/prisma.service';**

**import { User } from '@prisma/client';**

**@Injectable()**

**export class UsersRepository {**

**constructor(private prisma: PrismaService) {}**

**async getAll() {**

**return this.prisma.user.findMany();**

**}**

**async getById(id: number) {**

**return this.prisma.user.findUnique({ where: { id } });**

```
}
```

```
 async create(userData: Prisma.UserCreateInput):
 Promise<User> {

 return this.prisma.user.create({ data: userData });

 }
```

```
 async getByEmail(email: string){

 return this.prisma.user.findUnique({ where: { email } });

 }
```

} Prisma no se reconoce, no encuentra el nombre

El error ocurre porque no has importado el tipo `Prisma` desde el cliente generado por Prisma. Aquí está la solución completa:

## Solución para el error "Cannot find name 'Prisma'"

1. **Importa el namespace Prisma** desde `@prisma/client`:

```
typescript
// src/users/repositories/users.repository.ts
import { Injectable } from '@nestjs/common';
import { PrismaService } from '../../../prisma/prisma.service';
import { User, Prisma } from '@prisma/client'; // Importa Prisma aquí

@Injectable()
export class UsersRepository {
 constructor(private prisma: PrismaService) {}

 async getAll(): Promise<User[]> {
 return this.prisma.user.findMany();
 }

 async getById(id: number): Promise<User | null> {
 return this.prisma.user.findUnique({ where: { id } });
 }

 async create(userData: Prisma.UserCreateInput): Promise<User> {
```

```

 return this.prisma.user.create({ data: userData });
 }

 async getByEmail(email: string): Promise<User | null> {
 return this.prisma.user.findUnique({ where: { email } });
 }
}

```

## Verificación adicional:

### 1. Asegúrate que hayas generado el cliente Prisma:

```
bash
```

- `npx prisma generate`

### • Revisa tu `schema.prisma`:

```

prisma
2. generator client {
3. provider = "prisma-client-js"
4. }

```

### 5. Estructura de imports correcta:

- `User` → Tipo para entidades completas
- `Prisma` → Namespace con tipos para operaciones (`CreateInput`, `WhereInput`, etc.)

## Tipos útiles de Prisma que puedes necesitar:

| Tipo                                     | Descripción                               | Ejemplo de uso                                              |
|------------------------------------------|-------------------------------------------|-------------------------------------------------------------|
| <code>Prisma.UserCreateInput</code>      | Tipo para operaciones <code>create</code> | <code>create(userData: Prisma.UserCreateInput)</code>       |
| <code>Prisma.UserWhereUniqueInput</code> | Tipo para condiciones únicas              | <code>findUnique(where: Prisma.UserWhereUniqueInput)</code> |
| <code>Prisma.UserUpdateInput</code>      | Tipo para operaciones <code>update</code> | <code>update(data: Prisma.UserUpdateInput)</code>           |

## Si el error persiste:

### 1. Borra y regenera los tipos:

```
bash
```

- `rm -rf node_modules/.prisma`  
`npx prisma generate`

- **Reinicia tu servidor de desarrollo** (si estás usando `nest start --watch`)
- **Verifica la instalación de Prisma:**

bash

```
3. npm list @prisma/client
```

¿Necesitas que explique más sobre los tipos avanzados de Prisma o cómo estructurar operaciones más complejas? ☐

¿Necesitas ayuda con algún mensaje de error específico que aparezca después de estos pasos? ☐

Usuario\Desktop\Proyecto

```
A3SATEL\Backend\3tt\node_modules\.prisma\client\default.js:43 throw new
Error('@prisma/client did not initialize yet. Please run "prisma generate" and try to
import it again.');
```

^ Error: @prisma/client did not initialize yet. Please run "prisma generate" and try to import it again. at new PrismaClient (C:\Users\Usuario\Desktop\Proyecto A3SATEL\Backend\3tt\node\_modules\.prisma\client\default.js:43:11) at Object.<anonymous> (C:\Users\Usuario\Desktop\Proyecto A3SATEL\Backend\3tt\test-connection.ts:4:16) at Module.\_compile (node:internal/modules/cjs/loader:1469:14) at Module.m.\_compile (C:\Users\Usuario\Desktop\Proyecto A3SATEL\Backend\3tt\node\_modules\typescript\src\index.ts:1618:23) at Module.\_extensions.js (node:internal/modules/cjs/loader:1548:10) at Object.require.extensions.<computed> [as .ts] (C:\Users\Usuario\Desktop\Proyecto A3SATEL\Backend\3tt\node\_modules\typescript\src\index.ts:1621:12) at Module.load (node:internal/modules/cjs/loader:1288:32) at Function.Module.\_load (node:internal/modules/cjs/loader:1104:12) at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run\_main:174:12) at phase4 (C:\Users\Usuario\Desktop\Proyecto A3SATEL\Backend\3tt\node\_modules\typescript\src\bin.ts:649:14) PS C:\Users\Usuario\Desktop\Proyecto A3SATEL\Backend\3tt> npx prisma db execute --file .\prisma\test.sql --schema .\prisma\schema.prisma Error: Provided --file at .\prisma\test.sql doesn't exist. PS C:\Users\Usuario\Desktop\Proyecto A3SATEL\Backend\3tt>

## Solución Definitiva para el Error de Prisma Client en Windows

Vamos a resolver este problema de una vez por todas con una solución completa:

### 1. Solución Paso a Paso

## 1.1 Limpieza completa

powershell

```
Elimina todos los archivos generados por Prisma
Remove-Item -Recurse -Force node_modules\.prisma
Remove-Item -Recurse -Force node_modules\@prisma
Remove-Item -Force .\generated\prisma -ErrorAction SilentlyContinue
```

# Reinstala dependencias

```
npm uninstall @prisma/client prisma
npm install @prisma/client@latest prisma@latest
```

## 1.2 Configuración correcta del esquema

Edita tu `prisma/schema.prisma` para asegurarte que tenga:

prisma

```
generator client {
 provider = "prisma-client-js"
 output = "../node_modules/.prisma/client"
}

datasource db {
 provider = "oracle"
 url = env("DATABASE_URL")
 relationMode = "prisma"
}
```

## 1.3 Generación del cliente

powershell

```
npx prisma generate --force
```

# 2. Creación del archivo de prueba SQL

## 2.1 Crea el archivo test.sql

powershell

```
New-Item -Path .\prisma\test.sql -ItemType File -Value "SELECT * FROM
USERS;"
```

## 2.2 Ejecuta la prueba directa

powershell

```
npx prisma db execute --file .\prisma\test.sql --schema
.\prisma/schema.prisma
```

# 3. Solución Alternativa con TypeScript

## 3.1 Crea un nuevo archivo de prueba

powershell

```
New-Item -Path .\prisma-test.ts -ItemType File
```

## 3.2 Contenido de prisma-test.ts

typescript

```
import { PrismaClient } from '@prisma/client'

const prisma = new PrismaClient({
 log: ['query', 'info', 'warn', 'error']
})

async function main() {
 console.log('Testing Oracle connection...')

 // Prueba 1: Consulta directa
 const result = await prisma.$queryRaw`SELECT 1 as test FROM DUAL`
 console.log('Basic query result:', result)

 // Prueba 2: Consulta a tu tabla
 try {
 const users = await prisma.users.findMany()
 console.log('Users:', users)
 } catch (error) {
 console.error('Error querying users:', error)
 }
}

main()
 .catch(e => console.error('Error:', e))
 .finally(() => prisma.$disconnect())
```

## 3.3 Ejecuta la prueba

powershell

```
npx ts-node .\prisma-test.ts
```

# 4. Verificaciones Finales

### 1. Estructura del proyecto:

text

- a3tt/
  - ├─ node\_modules/
  - ├─ prisma/
    - ├─ schema.prisma
    - └─ test.sql
  - ├─ prisma-test.ts
  - ├─ package.json
  - └─ tsconfig.json

### • Variables de entorno:

- Verifica que tu .env tenga:

text

- •

```
DATABASE_URL="oracle://A3TT:A3TTsatel@localhost:1521/XE"
```

- **Permisos de Oracle:**

- Asegúrate que el usuario A3TT tenga permisos:

```
sql
```

```
3.
```

- GRANT CONNECT, RESOURCE TO A3TT;
- GRANT SELECT, INSERT, UPDATE, DELETE ON USERS TO A3TT;

## 5. Si el error persiste

### 1. Actualiza todo:

```
powershell
```

- npm update --force
- ```
npx prisma generate --force
```

- **Prueba con Node directamente:**

```
powershell
```

- node -e "require('@prisma/client')"

- **Verifica la instalación:**

```
powershell
```

```
3. npm list @prisma/client --depth=0
```

¿Necesitas ayuda con algún mensaje específico que aparezca después de estos pasos? ☐

El error indica que **Prisma no soporta Oracle Database de forma nativa**. Los providers oficiales de Prisma son: postgresql, mysql, sqlite, sqlserver, mongodb y cockroachdb.

Para trabajar con Oracle en NestJS, tienes estas opciones:

Opción 1: Usar TypeORM (Recomendado)

TypeORM tiene soporte completo para Oracle. Te ayudo a configurarlo:

Opción 2: Driver Nativo Oracle (Sin ORM)

Si prefieres usar el driver nativo:

Instalación de Dependencias

Para **TypeORM**:

```
npm install @nestjs/typeorm typeorm oracledb  
npm install @nestjs/config
```

Para **Driver Nativo**:

```
npm install oracledb  
npm install @nestjs/config
```

Recomendación

Te recomiendo usar **TypeORM** porque:

- ☐ Soporte completo para Oracle
- ☐ Migrations automáticas
- ☐ Type safety
- ☐ Mejor integración con NestJS
- ☐ Más fácil de mantener

¿Cuál opción prefieres que implementemos?