Full Stack Development BootCamp - Day 4

Build Real-World Apps from Backend to Frontend!

Trainer:

Muhammad Maaz Sheikh
Technical Team Lead

Tech Stack Covered:

- Nest Js
- PostgreSQL
- React Js
- Real world project development

Organized By:

Iqra University

Recap Day 3

Table of Content

Introduction to ORM

Introduction to ORM

What is ORM ?

Object-Relational Mapping (ORM) simplifies database interactions by allowing developers to use objects from their programming language to interact with relational databases (like SQL) instead of writing raw SQL queries. ORMs act as a bridge, mapping object-oriented models to relational database structures, offering advantages like increased code readability, maintainability, and reduced boilerplate. In NestJS, which is a framework for building server-side applications with Node.js and TypeScript.

What use ORMs ?

➔ Simplified Database Interactions
➔ Code Readability and Maintainability
➔ Data Type Consistency
➔ Improved Scalability
➔ It can be used with both relational databases like mysql, oracle, postgresql, maria db and nosql like mongodb

What is TypeORM?

Introduction to TypeORM

What is TypeORM ?

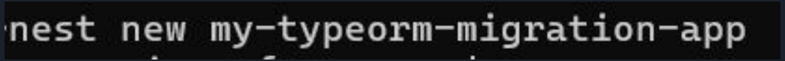
TypeORM is an ORM that can run in NodeJS, Browser, Cordova, PhoneGap, Ionic, React Native, NativeScript, Expo, and Electron platforms and can be used with TypeScript and JavaScript (ES5, ES6, ES7, ES8). Its goal is to always support the latest JavaScript features and provide additional features that help you to develop any kind of application that uses databases - from small applications with a few tables to large-scale enterprise applications with multiple databases.

TypeORM is highly influenced by other ORMs, such as **Hibernate**, **Doctrine** and **Entity Framework**.

Installing & Configuring Type ORM in Nest JS

Installing TypeORM

Create a New Nest JS Project

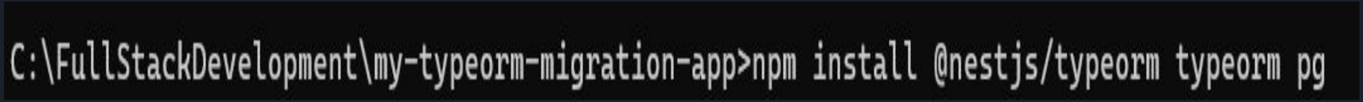


```
nest new my-typeorm-migration-app
```

- ➔ Choose npm when prompted.
- ➔ This creates a full Nest JS starter project.
- ➔ After creating project come in the project directory and install typeOrm and database driver

Install TypeORM and a database driver

For PostgreSQL Db:

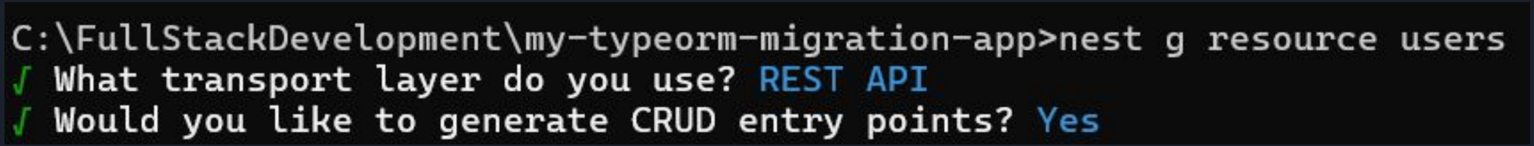


```
C:\FullStackDevelopment\my-typeorm-migration-app>npm install @nestjs/typeorm typeorm pg
```

Defining Models/Entities with TypeORM

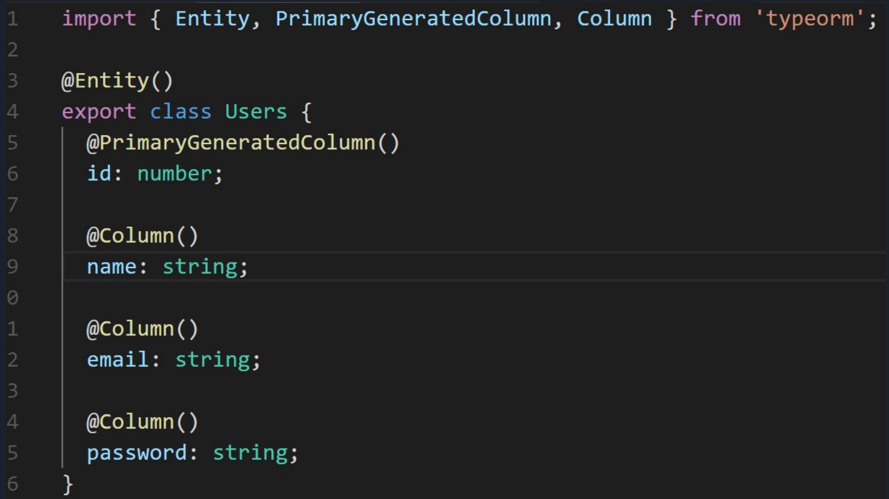
Create Module, Controller, Service, Dto and Entity:

➔ Create all this files at once with one below command:



```
C:\FullStackDevelopment\my-typeorm-migration-app>nest g resource users
√ What transport layer do you use? REST API
√ Would you like to generate CRUD entry points? Yes
```

➔ Then open a **user.entity.ts file** in **src/users/entities** and write a below code

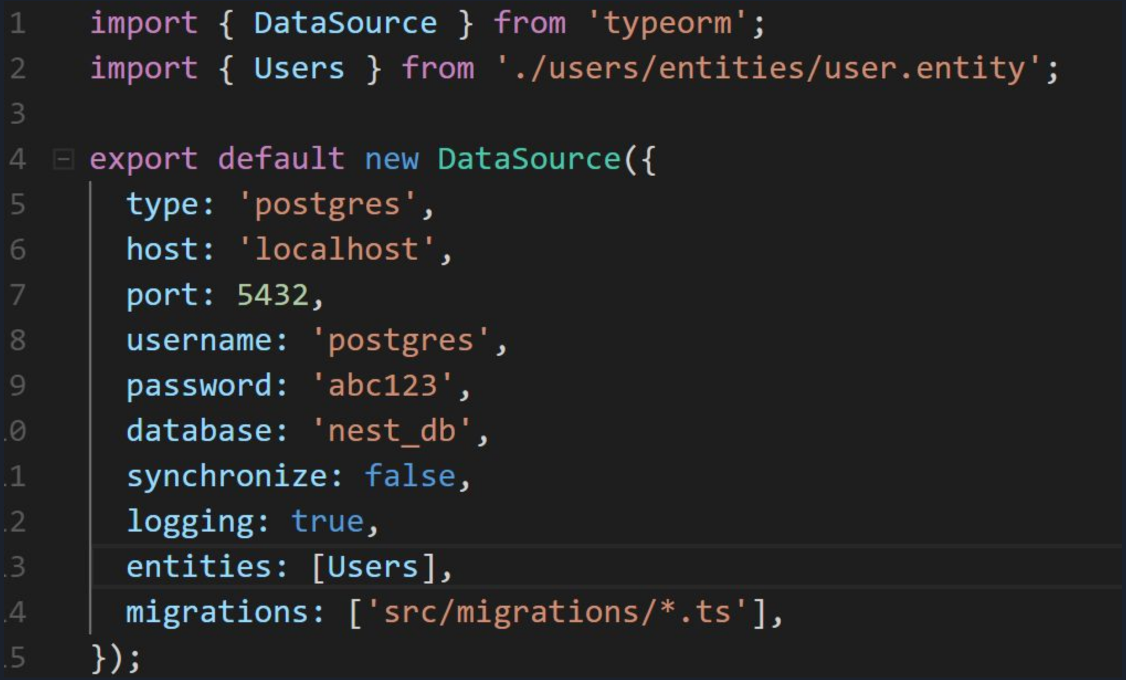


```
import { Entity, PrimaryGeneratedColumn, Column } from 'typeorm';

@Entity()
export class Users {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  name: string;

  @Column()
  email: string;

  @Column()
  password: string;
}
```

Installing & Configuring TypeORM

Configure TypeORM with Migrations:

- ➔ Create **src/data-source.ts** in **src:**



```
import { DataSource } from 'typeorm';
import { Users } from './users/entities/user.entity';

export default new DataSource({
  type: 'postgres',
  host: 'localhost',
  port: 5432,
  username: 'postgres',
  password: 'abc123',
  database: 'nest_db',
  synchronize: false,
  logging: true,
  entities: [Users],
  migrations: ['src/migrations/*.ts'],
});
```

Installing & Configuring TypeORM

Configure TypeORM in App Module:

- ➔ Use **TypeOrmModule.forRoot**() in **AppModule**
- ➔ In **src/app.module.ts** add below line of code in **Module:**
- ➔ Import **-> import { TypeOrmModule } from '@nestjs/typeorm';**

```
imports: [
    TypeOrmModule.forRoot({
      type: 'postgres',
      host: 'localhost',
      port: 5432,
      username: 'postgres',
      password: 'yourpassword',
      database: 'yourdb',
      entities: [Users], // ✅ include all your entities
      synchronize: false, // ✅ set to false when using migrations
    }),
    UsersModule,
  ],
```

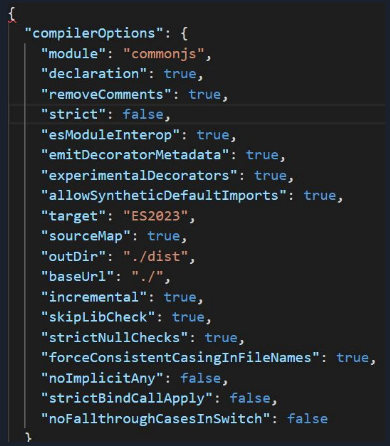
Installing & Configuring TypeORM

Configure TypeORM with Migrations:

➔ Update **Package.json** file in **script** section add below lines for migrations:

```
  "typeorm": "ts-node -r tsconfig-paths/register ./node_modules/typeorm/cli.js",
  "migration:generate": "npm run typeorm -- migration:generate",
  "migration:run": "npm run typeorm -- migration:run --dataSource src/data-source.ts",
  "migration:revert": "npm run typeorm -- migration:revert --dataSource src/data-source.ts"
```

➔ Update or Configure **tsconfig.json** file.



```
{
  "compilerOptions": {
    "module": "commonjs",
    "declaration": true,
    "removeComments": true,
    "strict": false,
    "esModuleInterop": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "allowSyntheticDefaultImports": true,
    "target": "ES2023",
    "sourceMap": true,
    "outDir": "./dist",
    "baseUrl": "./",
    "incremental": true,
    "skipLibCheck": true,
    "strictNullChecks": true,
    "forceConsistentCasingInFileNames": true,
    "noImplicitAny": false,
    "strictBindCallApply": false,
    "noFallthroughCasesInSwitch": false
  }
```

Create & Run Database Migrations

Generate and Run Migration:

➔ Generate migration by using below command:

```
  npm run migration:generate src/migrations/CreateUserTable -- --dataSource
src/data-source.ts
```

➔ Run migration and apply changes to the db by using below command

```
npm run migration:run
```

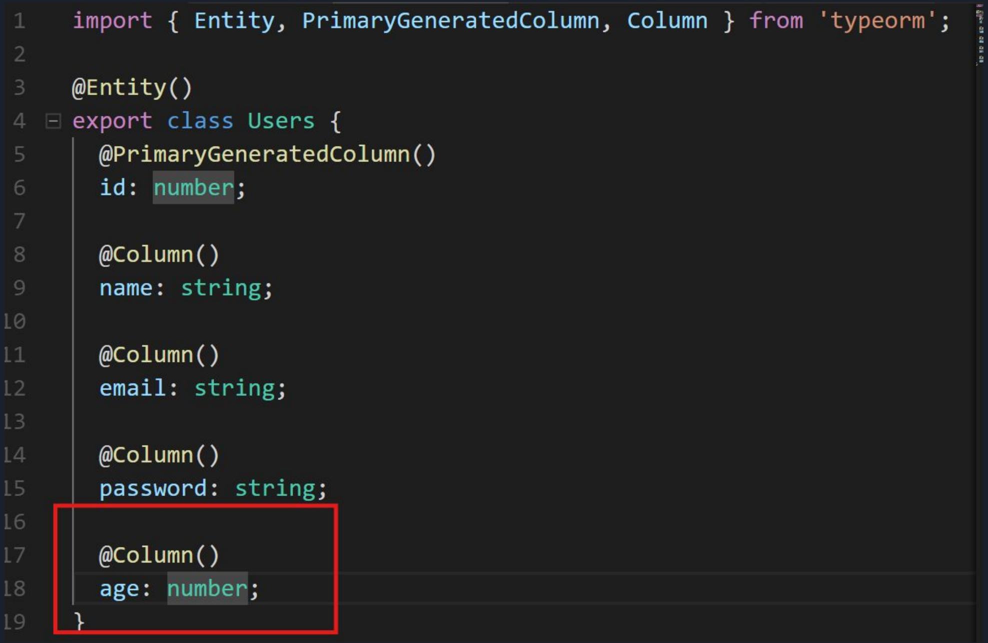
➔ You'll see SQL queries logged in the terminal and the user table created in your database.

➔ Use any tool (pgAdmin, DBeaver, psql) and run: **SELECT * FROM users;**

Update Entity and Run Migration Again

Update User Entity and Run Migration:

- ➔ Let’s say you want to add a **age** column to your **User** entity.
- ➔ Update **src/user/user.entity.ts**:



```typescript
import { Entity, PrimaryGeneratedColumn, Column } from 'typeorm';

@Entity()
export class Users {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  name: string;

  @Column()
  email: string;

  @Column()
  password: string;

  @Column()
  age: number;
}
```

Update Entity and Run Migration Again

Update User Entity and Run Migration:

Generate New Migration

- ➔ Generate a new migration by using below command:

```
 npm run migration:generate src/migrations/AddAgeColumn -- --dataSource
src/data-source.ts
```

- ➔ It will detect the new column and generate SQL

Run Migration

- ➔ Run migration and apply changes to the db by using below command

  ```
  npm run migration:run
  ```

- ➔ Use any tool (pgAdmin, DBeaver, psql) and run: **SELECT * FROM "user";**

Revert Migration

Run Revert Migration:

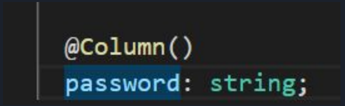
➔ To undo the last migration run below command:

```
npm run migration:revert
```

➔ Above command will revert the last migration from database in which we added the **age** column in db.

➔ Now manually delete the migration file from **migrations** folder.

➔ Now add new column in User Entity named it ‘**Password’.** We will use it later.



```
@Column()
password: string;
```

➔ Now again create and run the new migration

➔ Create Migration
  - `npm run migration:generate src/migrations/AddPasswordColumn -- --dataSource src/data-source.ts`

➔ Run Migration
  - `npm run migration:run`

CRUD Operations with TypeORM

CRUD Operations with TypeORM

Setup User.Service:

Step#01: Open already added user service and install below dependencies.

npm install @nestjs/jwt bcrypt

npm install @types/bcrypt

Step#02: Define imports at the top of the service file

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { Users } from './entities/user.entity';
import { CreateUserDto } from './dto/create-user.dto';
import { UpdateUserDto } from './dto/update-user.dto';
import * as bcrypt from 'bcrypt';
```

CRUD Operations with TypeORM

Step#03: Define Constructor in service class as define below.

```
constructor(
   @InjectRepository(Users)
   private readonly userRepository: Repository<Users>,
 ) {}
```

Step#04: Now define the **GET** method in service to retrieve the data from the database

```
async findAll(): Promise<Users[]> {
   return this.userRepository.find();
 }
```

Step#05: Now define the **GET By ID** method in service to retrieve the data from the database by Id.

```
 async findOne(id: number): Promise<Users | null> {
   return this.userRepository.findOneBy({ id });
 }
```

CRUD Operations with TypeORM

Step#06: Now define the **Create** method in service to create the new user data in the database and install below dependencies before writing the create method.

```
async create(createUserDto: CreateUserDto): Promise<Users> {
   const hashedPassword = await bcrypt.hash(createUserDto.password, 10);
   const user = this.userRepository.create({
     ...createUserDto,
     password: hashedPassword
   });
   return this.userRepository.save(user);
 }
```

Step#07: Now define the **Update** method in service to save the data from the database by Id.

```
async update(id: number, updateUserDto: UpdateUserDto): Promise<Users | null> {
   const user = await this.userRepository.findOneBy({ id });
   if (!user) return null;
   const updatedUserData = { ...updateUserDto };
   if (updateUserDto.password) {
     updatedUserData.password = await bcrypt.hash(updateUserDto.password, 10);
   } else {
     delete updatedUserData.password; // Avoid accidentally setting password to undefined
   }
   const updatedUser = this.userRepository.merge(user, updatedUserData);
   return this.userRepository.save(updatedUser);
 }
```

CRUD Operations with TypeORM

Step#08: Now define the **Remove By ID** method in service to remove wwthe new data in the database

```
async remove(id: number): Promise<Users | null> {
   const user = await this.userRepository.findOneBy({ id });
   if (!user) return null;

   await this.userRepository.remove(user);
   return user;
 }
```

CRUD Operations with TypeORM

Setup User.Controller:

Step#01: Define imports at the top of the controller file

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

CRUD Operations with TypeORM

Setup User.Controller:

Step#02: Define constructor in the **UsersController** class and inject **UsersService** in it.

```
constructor(private readonly usersService: UsersService) {}
```

Step#03: Now define the **GET** methods in controller to retrieve all data and data by Id from the database

```
 @Get()
  async findAll(): Promise<Users[]> {
    return this.usersService.findAll();
  }

  @Get(':id')
  async findOne(@Param('id', ParseIntPipe) id: number): Promise<Users>{
    const user = await this.usersService.findOne(id);
    if(!user) throw new NotFoundException('User not found');
    return user;
  }
```

CRUD Operations with TypeORM

Setup User.Controller:

Step#04: Now define the **Post** method in controller to create data in database

```
@Post()
async create(@Body() createUserDto: CreateUserDto): Promise<Users> {
  return this.usersService.create(createUserDto);
}
```

Step#05: Now define the **Put** method in controller to update data in database

```
@Put(':id')
@HttpCode(HttpStatus.OK)
async update(
  @Param('id', ParseIntPipe) id: number,
  @Body() updateUserDto: UpdateUserDto,
): Promise<{ message: string }> {
  const updated = await this.usersService.update(id, updateUserDto);
  if (!updated) throw new NotFoundException('User not found for update');
  return { message: 'Record updated successfully!' };
}
```

CRUD Operations with TypeORM

Setup User.Controller:

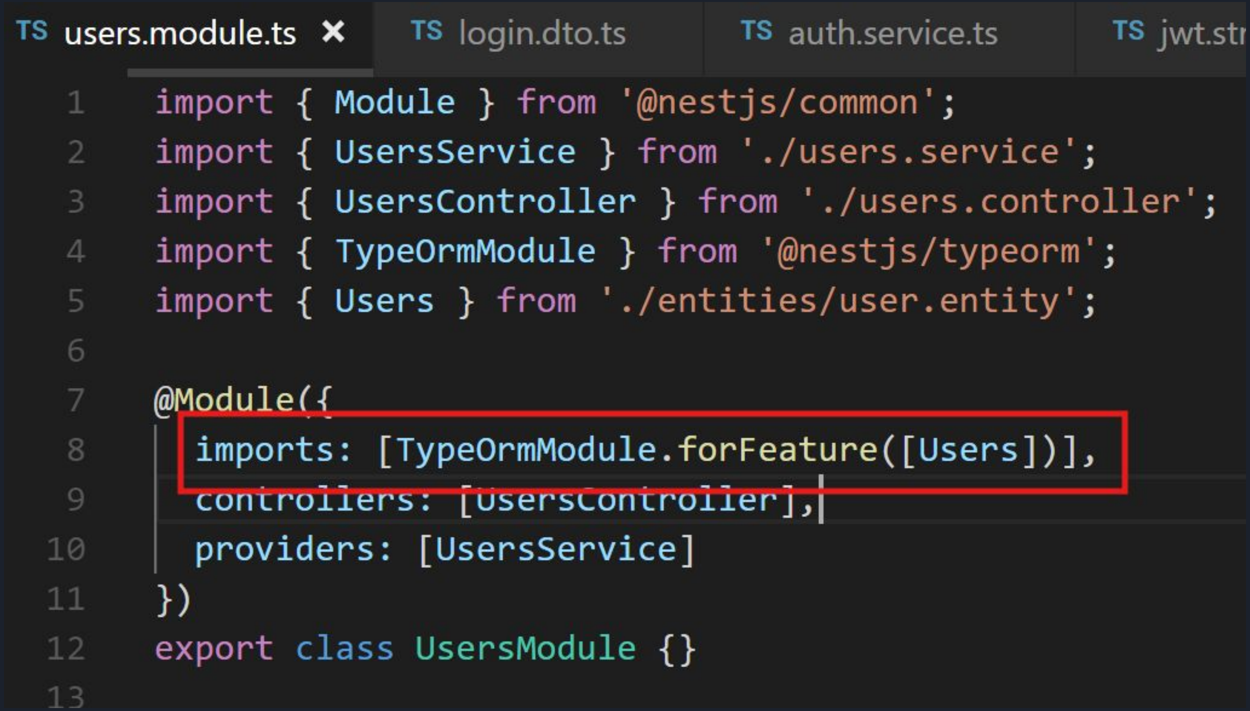
Step#06: Now define the **Delete** method in controller to delete data by Id from the database

```
  @Delete(':id')
 async remove(@Param('id', ParseIntPipe) id: number): Promise<Users> {
   const deleted = await this.usersService.remove(id);
   if (!deleted) throw new NotFoundException('User not found for deletion');
   return deleted;
 }
```

CRUD Operations with TypeORM

Setup User.Module.ts:

In your **User.Module.ts** or wherever your UserService is declared add this:



```
import { Module } from '@nestjs/common';
import { UsersService } from './users.service';
import { UsersController } from './users.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { Users } from './entities/user.entity';

@Module({
  imports: [TypeOrmModule.forFeature([Users])],
  controllers: [UsersController],
  providers: [UsersService]
})
export class UsersModule {}

```

CRUD Operations with TypeORM

Setup create-user.dto.ts:

Install Class-Validator: First, add **class-validator** and **class-transformer** to your project:

```
npm install class-validator class-transformer
```

open **create-user.dto.ts** file in **users/dto** and add below code

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

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

   @IsEmail()
   email: string;

   @IsString()
   @MinLength(6)
   @MaxLength(20)
   password: string;
 }
```

CRUD Operations with TypeORM

Setup update-user.dto.ts:

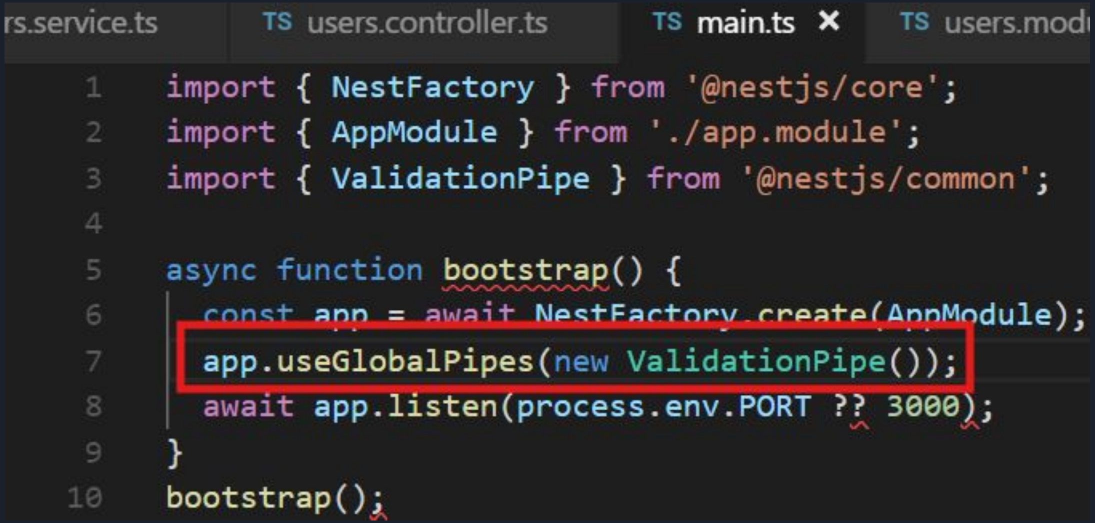
open **update-user.dto.ts** file in **users/dto** and add below code

```
import { IsEmail, IsOptional, IsString, MinLength, MaxLength, IsInt } from 'class-validator';

export class UpdateUserDto {
 @IsOptional()
 @IsString()
 name?: string;

 @IsOptional()
 @IsEmail()
 email?: string;

 @IsOptional()
 @IsString()
 @MinLength(6)
 @MaxLength(20)
 password?: string;
}
```

CRUD Operations with TypeORM

Enable Validation Globally

open **main.ts** file and define validation globally as below



rs.service.ts | TS users.controller.ts | TS main.ts × | TS users.mod

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { ValidationPipe } from '@nestjs/common';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe());
  await app.listen(process.env.PORT ?? 3000);
}
bootstrap();
```

Test API Endpoints

- Start the Server
  - Npm run start
- GET Request (Get All Users)
  - **(GET)** http://localhost:3000/users
- GET Request (Get User By Id)
  - **(GET)** http://localhost:3000/users/1
- POST Request (Create a New User)
  - **(POST)** http://localhost:3000/users
  - Body: { "name": "Bob", "email": "bob@example.com" }
- PUT Request (Replace User)
  - **(PUT)** http://localhost:3000/users/1
  - Body: { "name": "John", "email": "john@example.com" }
- PATCH Request (Partially Update User)
  - **(PATCH)** http://localhost:3000/users/1
  - Body: { "email": "john.doe@example.com" }
- DELETE Request
  - **(DELETE)** http://localhost:3000/users/1

Authentication & Authorization via JWT

What is Authentication & Authorization ?

Authentication & Authorization

When building secure applications, understanding the distinction between **authentication** and **authorization** is crucial. **Authentication** verifies a user's identity, while **authorization** determines what resources a user can access. In this section, we'll explore these concepts and demonstrate how to implement them in a Nest JS application with Postgres to secure our endpoints.

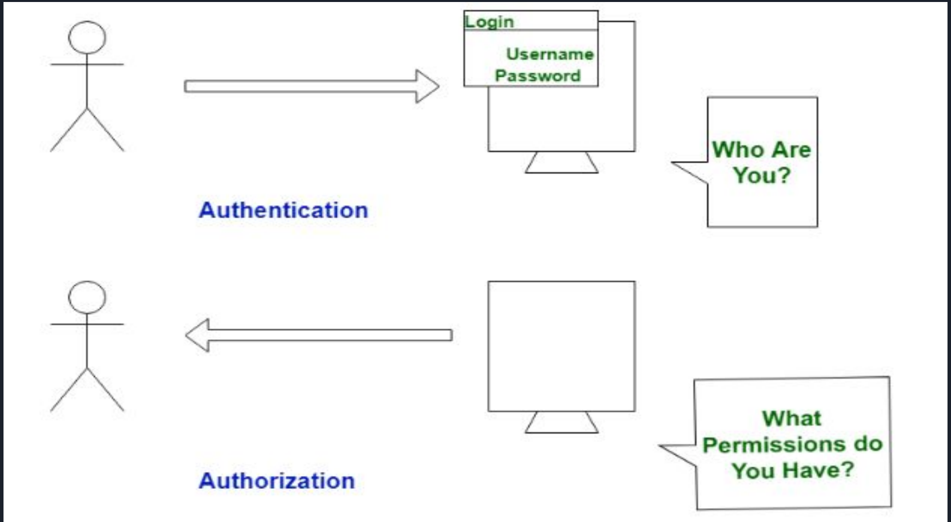
What is Authentication?

Authentication is the method of verifying the identity of a consumer or system to ensure they're who they claim to be. It involves checking credentials which include usernames, passwords, or biometric information like fingerprints or facial recognition. This step is vital for securing access to systems, programs, and sensitive records. By confirming identities, authentication saves you from unauthorized entry and protects you against safety breaches.

Authentication & Authorization

What is Authorization?

Authorization is the method of figuring out and granting permissions to a demonstrated user or system, specifying what assets they can access and what actions they're allowed to carry out. It comes after authentication and guarantees that the authenticated entity has the proper rights to use certain data, applications, or services. This step is important for implementing protection guidelines and controlling access within the system, thereby stopping unauthorized activities.



JWT Authentication

What Is JWT Authentication?

JSON Web Token (JWT) authentication is a stateless method of securely transmitting information between parties as a JavaScript Object Notation (JSON) object. It is often used to authenticate and authorize users in web applications and APIs.

In the authentication world, “stateless” means a mechanism in which the server does not maintain any session state between requests. In a stateless authentication system, each request is self-contained and includes all the necessary information to authenticate and authorize the user or entity. In the case of JWT authentication, this comes in the form of a token.

A JSON token consists of three parts:

- A Header containing information about the type of token and algorithms used to generate the signature.
- A Payload containing the “claims” (ID and authentication verifications) made by the user that can include a User ID, the user’s name, an email address, and meta information about the operation of the token.
- A Signature, or cryptographic mechanism, is used to verify the token’s integrity.

JWT Authentication

What Is JWT Authentication? (cont.)

Together, the header, payload, and signature make up the JSON Web Token, typically passed between the client and the server in the HTTP Authorization header or in the body of an HTTP request or response. The server can then verify the signature to ensure that the token is valid and has not been modified and use the information in the payload to authenticate the user.

Here's how JWT authentication works:

- **User Login**: The user provides their credentials (such as a username and password) to the web application or system for verification, which is transmitted to the authentication server.
- **Token Generation**: Upon successful authentication, the server generates a JSON token containing critical information about the user and the authentication session. The server sends the token to the client for verification.
- **Token Storage:** The client stores the token, usually in a cookie or purpose-marked local storage, and includes it in subsequent requests to the server.
- **User Verification**: When the client sends a request to the application server, it verifies the signature in the token and checks the claims in the payload to ensure that the user can access the requested resource.
- **Server Response**: If the JWT is valid and the user can access the requested resource.
- **Token Expiration**: When the JWT expires, the client must obtain a new JWT by logging in again.

Implementation Authentication via Jwt Auth

Implement JWT Step by Step for Auth

Step# 01: Install Required Packages/Dependencies:

- ➔ npm install @nestjs/passport passport passport-jwt
- ➔ npm install --save-dev @types/passport-jwt
- ➔ npm install passport-local
- ➔ npm install -D @types/passport-local

Step# 02: Create Auth Module, Controller, Service:

- ➔ nest g module auth
- ➔ nest g service auth
- ➔ nest g controller auth

Step# 03: Create Auth Dto for perform Login:

- ➔ Create a **dto** folder in **Auth** folder **src/auth/dto**
- ➔ Create **login.dto.ts** and use below code:

```
export class LoginDto {
  email: string;
  password: string;
}
```

Implement JWT Step by Step for Auth

Step# 04: Add Auth Logic in Service:

- ➔ In **auth.service.ts**:
- ➔ npm install --save-dev @types/passport-jwt

Imports:

```
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { JwtService } from '@nestjs/jwt';
import { UsersService } from '../users/users.service';
import * as bcrypt from 'bcrypt';
```

Imports Explanation:

- ➔ **JwtService** -> This will authenticate the request
- ➔ **UnauthorizedException** -> This exception will fire in case if some one is unauthorize
- ➔ **UsersService** -> This was the service that we created earlier for creating the user
- ➔ **bcrypt** -> This library will encrypt/hash the user provided password to unreadable format

Implement JWT Step by Step for Auth

Auth Service code:

```
@Injectable()
export class AuthService {
 constructor(
   private usersService: UsersService,
   private jwtService: JwtService,
 ) {}

 async validateUser(email: string, pass: string): Promise<any> {
   const user = await this.usersService.findByEmail(email);
   if (user && await bcrypt.compare(pass, user.password)) {
     const { password, ...result } = user;
     return result;
   }
   return null;
 }

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

Implement JWT Step by Step for Auth

Step# 05: Setup Local & Jwt Strategy:

- ➔ Create a **strategies** folder in auth folder **src/auth/strategies**:
- ➔ Create two files manually in **strategies** folder with name **jwt.strategy.ts** and **local.strategy.ts**.

Import for local.strategy.ts:

```
import { Strategy } from 'passport-local';
import { PassportStrategy } from '@nestjs/passport';
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { AuthService } from '../auth.service';
```

Import for jwt.strategy.ts:

```
import { Injectable } from '@nestjs/common';
import { PassportStrategy } from '@nestjs/passport';
import { ExtractJwt, Strategy } from 'passport-jwt';
```

Implement JWT Step by Step for Auth

local.strategy.ts:

```
@Injectable()
export class LocalStrategy extends PassportStrategy(Strategy) {
 constructor(private authService: AuthService) {
   super({ usernameField: 'email' }); // use email instead of username
 }

 async validate(email: string, password: string): Promise<any> {
   const user = await this.authService.validateUser(email, password);
   if (!user) throw new UnauthorizedException();
   return user;
 }
}
```

Implement JWT Step by Step for Auth

jwt.strategy.ts:

```
@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
 constructor() {
   super({
     jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
     ignoreExpiration: false,
     secretOrKey: 'your_secret_key', // Use env var in real apps
   });
 }

 async validate(payload: any) {
   return { userId: payload.sub, email: payload.username, role: payload.role };
 }
}
```

Implement JWT Step by Step for Auth

Step# 06: Configure JWT in Auth Module

```
import { Module } from '@nestjs/common';
import { AuthController } from './auth.controller';
import { AuthService } from './auth.service';
import { JwtStrategy } from './strategies/jwt.strategy';
import { LocalStrategy } from './strategies/local.strategy';
import { UsersModule } from '../users/users.module';
import { PassportModule } from '@nestjs/passport';
import { JwtModule, JwtService } from '@nestjs/jwt';

@Module({
 imports: [
   UsersModule,
   PassportModule,
   JwtModule.register({
     secret: 'your_secret_key',
     signOptions: { expiresIn: '1h' },
   }),
 ],
 controllers: [AuthController],
 providers: [AuthService, LocalStrategy, JwtStrategy]
})
export class AuthModule {}
```

Implement JWT Step by Step for Auth

Step# 07: Configure Guards

Add **guard** folder in **src/auth/guard** and add two files with name **jwt-auth.guard.ts** and **local-auth.guard.ts**

local-auth.guard.ts:

```
import { Injectable } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

@Injectable()
export class LocalAuthGuard extends AuthGuard('local') {}
```

jwt-auth.guard.ts:

```
import { Injectable } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {}
```

Implement JWT Step by Step for Auth

Step# 8: Auth Controller - Login Endpoint

```
import { Controller, Post, UseGuards, Request } from '@nestjs/common';
import { AuthService } from './auth.service';
import { LocalAuthGuard } from './guard/local-auth.guard';

@Controller('auth')
export class AuthController {

 constructor(private authService: AuthService)  {}

 @UseGuards(LocalAuthGuard)
 @Post('login')
 async login(@Request() req)  {
    return this.authService.login(req.user);
 }
}
```

Implement JWT Step by Step for Auth

Step# 09: Configure User Module to be used in Auth Module

users.module.ts:

```
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])],
 controllers: [UsersController],
 providers: [UsersService],
 exports: [UsersService], // 👈 this makes UsersService accessible to other modules
})
export class UsersModule {}
```

✅ What This Does

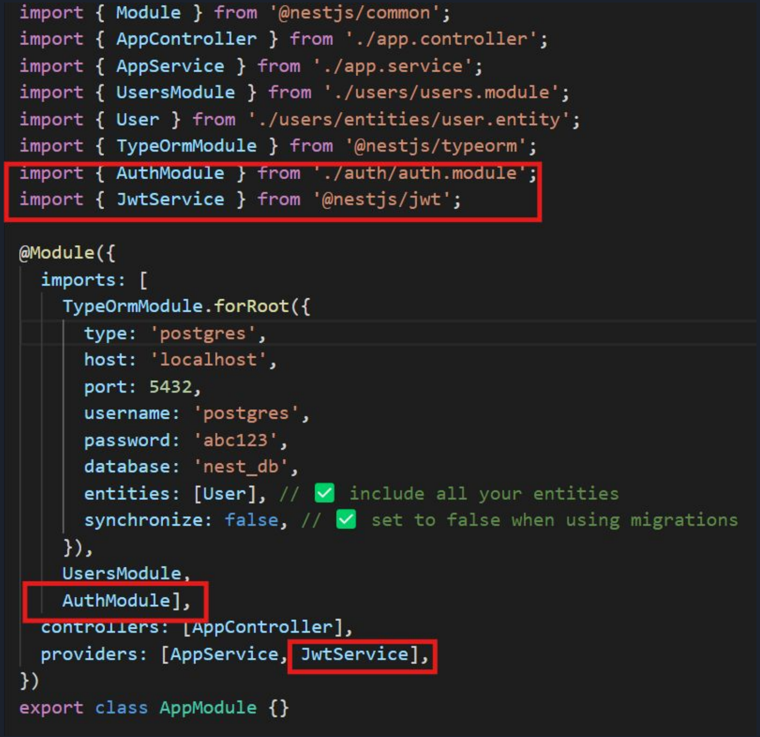
- Makes **UsersService** available to **AuthService**
- Resolves the **UnknownDependenciesException**.

Implement JWT Step by Step for Auth

Step# 10: Configure AppModule

Add imports and providers for the auth module and jwt service

app.module.ts:



```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { UsersModule } from './users/users.module';
import { User } from './users/entities/user.entity';
import { TypeOrmModule } from '@nestjs/typeorm';
import { AuthModule } from './auth/auth.module';
import { JwtService } from '@nestjs/jwt';

@Module({
  imports: [
    TypeOrmModule.forRoot({
      type: 'postgres',
      host: 'localhost',
      port: 5432,
      username: 'postgres',
      password: 'abc123',
      database: 'nest_db',
      entities: [User], // ✅ include all your entities
      synchronize: false, // ✅ set to false when using migrations
    }),
    UsersModule,
    AuthModule],
  controllers: [AppController],
  providers: [AppService, JwtService],
})
export class AppModule {}
```

Implement JWT Step by Step for Auth

Step# 11: Now include the findByEmail in User Service

- In **users.service.ts** add new method **findByEmail** as this method was called by the authservice to be authenticated when user perform login

```
async findByEmail(email: string): Promise<Users | null> {
    return this.userRepository.findOne({ where: { email } });
  }
```

This will validate and extract the data only with this email address

Implement JWT Step by Step for Auth

Step# 12: Protects Routes using JWT

In **your users.controller.ts**

users.controller.ts:

```
@UseGuards(JwtAuthGuard)  -> This line will protect the route
@Get()
findAll() {
  return this.usersService.findAll();
}
```

💡 **Quick Reminder**

- **@UseGuards(JwtAuthGuard)** this will enable and secure the api endpoint via jwt authentication
- **@Controller()** classes go in controllers: [...]
- **@Injectable()** services/strategies go in providers: [...]

Implement JWT Step by Step for Auth

Understanding of Guard and Strategies

➔ **Guards**: Protect Routes & Handle Authentication Flow

Guard	Purpose	How it Works
LocalAuthGuard	Used for login endpoint	Validates username/password via LocalStrategy, then attaches user to request object.
JwtAuthGuard	Used for protecting routes after login	Extracts token from Authorization: Bearer <token>, verifies with JwtStrategy, and lets user access protected resources.

➔ **Strategies**: Contain Logic for Auth Validation

Strategy	Purpose	How it Works
LocalStrategy	Handles logic for login credentials	Gets called by LocalAuthGuard. Uses validate(email, password) to check user.
JwtStrategy	Handles logic for token validation	Gets called by JwtAuthGuard. Validates and decodes the JWT payload.

Implement JWT Step by Step for Auth

Example Flow of Jwt and Local Strategy

🔐 **Login** (POST /auth/login)

- ➔ You use **@UseGuards(LocalAuthGuard)**
- ➔ Calls **LocalStrategy.validate()**
- ➔ If valid, **AuthService.login()** returns a JWT

🔒 **Protected route** (GET /user)

- ➔ You use **@UseGuards(JwtAuthGuard)**
- ➔ Calls **JwtStrategy.validate()**
- ➔ User extracted from token is attached to request