

# NestJS User Management Tutorial

## Building a Full-Stack User Management System with NestJS, PostgreSQL, TypeORM, and React [🔗](#)

Welcome to this step-by-step tutorial for building a full-stack user management system! We will use NestJS for our back-end (with PostgreSQL, TypeORM, Passport for authentication, class-validator for validation, and class-transformer for serialization), and React for our front-end. This tutorial is beginner-friendly, with all steps explained in the logical order of implementation.

### Table of Contents [🔗](#)

1. Introduction to the Stack
2. Setting Up the Backend with NestJS
  - a. Installing NestJS CLI
  - b. Creating a NestJS Project
  - c. Adding Dependencies
3. Configuring PostgreSQL & TypeORM
  - a. Database Creation
  - b. Global TypeORM Configuration
4. Creating the User Module
  - a. User Entity
  - b. User DTOs and Validation
  - c. User Service
  - d. User Controller
5. Using class-validator for Input Validation
6. Using class-transformer to Exclude Sensitive Fields
7. Setting Up Authentication with Passport
  - a. Auth Module & Service
  - b. Local Strategy (Email & Password Login)
  - c. JWT Strategy (Securing Protected Routes)
  - d. Auth Controller (Login & JWT issuance)
8. Implementing Registration & Profile Routes
  - a. /users/register Route
  - b. /users/profile Route (JWT Protected)
9. Building the React Frontend
  - a. Project Setup for React
  - b. Registration Form
  - c. Login Form
  - d. Profile Page (Fetching Protected Data)
  - e. Storing and Using the JWT Token
10. Conclusion & Next Steps

Let's dive in! 🚀

### Introduction to the Stack [🔗](#)

Before we start coding, let's briefly introduce the technologies we'll use and how they fit together:

- **NestJS:** A progressive Node.js framework for building efficient server-side applications. It provides a structured way to build scalable backends with TypeScript.
- **PostgreSQL:** A powerful open-source relational database. We'll use it to store our user data.
- **TypeORM:** An ORM (Object-Relational Mapping) library for TypeScript/JavaScript that integrates well with NestJS, making it easy to manage database tables as TypeScript classes (entities).
- **Passport:** A popular authentication middleware for Node.js. We will use Passport's Local Strategy for email/password login and JWT Strategy for protecting routes with JSON Web Tokens.
- **class-validator:** A validation library that integrates with NestJS to declaratively validate incoming request data (DTOs). It lets us add decorators like `@IsEmail()` or `@IsNotEmpty()` to DTO classes, automatically checking input at runtime.
- **class-transformer:** A library for transforming and sanitizing class instances. We will use it with NestJS's serialization interceptor to hide sensitive fields (like passwords) from the responses by using the `@Exclude()` decorator.
- **ReactJS:** Our front-end library of choice, which will interact with the NestJS backend through HTTP requests. We'll create a simple single-page application (SPA) with forms for registration and login, and a protected profile page.


### How they connect [↗](#)

When a user registers or logs in from the React frontend, the app will send HTTP requests to our NestJS API. The NestJS backend will validate and process these requests, interact with the PostgreSQL database via TypeORM, and return responses (like a JWT token or user info). React will then update the UI based on these responses (e.g., storing the JWT and showing the profile data).

With the overview out of the way, let's set up our backend!

## Setting Up the Backend with NestJS [↗](#)

### Creating NestJS Resources in Nx [↗](#)

 If you're using Nx, replace any Nest CLI resource generation like `nest g resource auth` with

```
1 npx nx g @nx/nest:resource backend/src/users/users --dry-run
```

Remove `--dry-run` to generate the files:

### Alternative: Installing NestJS CLI [↗](#)

First, ensure you have the NestJS CLI installed globally. The NestJS CLI helps scaffold projects and generate components like modules, services, and controllers.

If you haven't installed it yet, run:

```
1 npm install -g @nestjs/cli
```

This installs the Nest CLI globally on your system. You can verify the installation by running `nest --version`.

### Creating a NestJS Project [↗](#)

Next, create a new NestJS project. For this tutorial, we'll call it "user-auth-system":

```
1 nest new users
```

This command scaffolds a new NestJS project in a directory named `user-auth-system`. During setup, Nest may ask which package manager to use (npm or yarn)—choose your preference.

Once the command completes, navigate into the project directory:

```
1 cd users
```

You'll see a typical NestJS project structure (with `src/` containing `main.ts`, `app.module.ts`, etc.).

## Adding Dependencies [↗](#)

NestJS provides many features out-of-the-box. However, for our project, we need to install additional packages:


```
1 npm install @nestjs/typeorm typeorm pg \
2   @nestjs/passport passport passport-local passport-jwt \
3   bcrypt @nestjs/jwt \
4   class-validator class-transformer
5
```

Let's break these down:

- `@nestjs/typeorm` & `typeorm`: For connecting to PostgreSQL using TypeORM. The `pg` package is the PostgreSQL driver.
- `@nestjs/passport`, `passport`, `passport-local`, `passport-jwt`: Passport and its strategies for authentication. NestJS integrates with Passport through `@nestjs/passport`.
- `bcrypt`: A library for hashing passwords. It's a security best practice to never store passwords in plain text. We'll use `bcrypt` to hash passwords before saving to the database (and to verify during login).
- `@nestjs/jwt` & `jsonwebtoken`: Helps with generating and verifying JWTs in NestJS.
- `class-validator` & `class-transformer`: For validating incoming data and transforming outgoing data (like excluding password fields).

After running the install command, these will be added to your `package.json`.

## Configuring PostgreSQL & TypeORM [↗](#)

 In our project, we use `docker compose up -d` to run our database

### Database Creation [↗](#)

Make sure you have PostgreSQL running on your system and create a database for this project. You can use a tool like `psql` (PostgreSQL CLI) or a GUI tool. For example, using `psql`:

```
1 CREATE DATABASE user_auth_db;
```

This creates a new PostgreSQL database named `user_auth_db`. We'll use this database to store our users.

**Note:** Remember your PostgreSQL connection details (host, port, username, password, database name). By default, PostgreSQL runs on port 5432.

### Global TypeORM Configuration [↗](#)

We will configure TypeORM in the root module (`AppModule`) so it's available application-wide. Open the file `src/app.module.ts`. Modify it as follows to set up a global TypeORM connection:

```
1 import { Module } from '@nestjs/common';
2 import { TypeOrmModule } from '@nestjs/typeorm';
3 import { User } from './user/user.entity'; // (we'll create User entity soon)
4 import { UserModule } from './user/user.module'; // (we'll create User module soon)
5
6 @Module({
7   imports: [
8     TypeOrmModule.forRoot({
9       type: 'postgres',
10      host: 'localhost',
11      port: 5432,
```

```

12     username: 'your_db_username',
13     password: 'your_db_password',
14     database: 'user_auth_db',
15     entities: [User], // Specify all entities here, e.g., User entity
16     synchronize: true, // auto-create database schema (for dev only)
17   },
18   UserModule, // We'll add AuthModule later after creating it
19 ],
20 })
21 export class AppModule {}
22

```

#### Explanation:

- We import `TypeOrmModule.forRoot` and provide our database connection options. These include database type, host, port, credentials, and the database name.
- `entities: [User]` tells TypeORM which entities (tables) to sync with the database. We'll create the User entity in the next section.
- `synchronize: true` automatically syncs the entity schema with the database on startup (handy for development). Don't use `synchronize: true` in production as it may drop or alter tables unexpectedly; use migrations instead.
- We import our `UserModule` (to be created) so NestJS knows about our user-related functionality. We will also import `AuthModule` once we create it for authentication.

With this setup, when we run the NestJS application, it will connect to PostgreSQL and ensure the `user_auth_db` is ready to use.

## Creating the User Module [🔗](#)

The User Module will handle user-related operations, such as registering new users and retrieving user profiles. We will create:

- A User Entity (with TypeORM) to represent the users table
- User DTOs (Data Transfer Objects) to structure data for registration and responses
- User Service for business logic (like creating a user)
- User Controller for handling HTTP requests (register and profile routes)

To generate a module in NestJS, you can use the CLI, but we'll write the files manually here for clarity.

### User Entity [🔗](#)

Create a directory `src/user` (if not already created by CLI). Inside, create a file `user.entity.ts` for the User entity:

```

1 import { Entity, PrimaryGeneratedColumn, Column, CreateDateColumn } from 'typeorm';
2 import { Exclude } from 'class-transformer';
3
4 @Entity('users') // This decorator marks the class as a database table named 'users'
5 export class User {
6   @PrimaryGeneratedColumn()
7   id: number;
8
9   @Column({ unique: true })
10  email: string;
11
12  @Column()
13  @Exclude() // We use Exclude to prevent this field from being included when we return the user object
14  password: string;
15
16  @CreateDateColumn()
17  createdAt: Date;
18 }

```

**Explanation:**

- `@Entity('users')` : Marks the class as a database entity (table) named "users"
- `@PrimaryGeneratedColumn()` : An auto-incrementing primary key
- `@Column({ unique: true }) email` : A unique email column (we treat email as username)
- `@Column() password` : Column for password. We'll store the hashed password here
- `@Exclude()` : A decorator from class-transformer that will omit this field when transforming the entity to plain JSON. This helps ensure we never send passwords in API responses
- `@CreateDateColumn() createdAt` : Auto-managed timestamp when each user is created (just for info)

Now, update the `AppModule`'s `TypeOrmModule` import if needed to ensure the `User` entity is included. We already did `entities: [User]` in the previous step, which is why we imported `User` in `app.module.ts`.

**User DTOs and Validation** [🔗](#)

DTOs (Data Transfer Objects) define the shape of data for operations like registration. DTOs work hand-in-hand with class-validator decorators to enforce validation rules on incoming data.

Let's create two DTOs in `src/user`:

1. `create-user.dto.ts` for registration data
2. `user.dto.ts` for what data we send back (for example, exclude password)

`create-user.dto.ts`:

```
1 import { IsEmail, IsString, MinLength, IsNotEmpty } from 'class-validator';
2
3 export class CreateUserDto {
4   @IsEmail({}, { message: 'Invalid email' })
5   email: string;
6
7   @IsString()
8   @MinLength(6, { message: 'Password must be at least 6 characters long' })
9   password: string;
10 }
11
```

- `@IsEmail()` checks that the email is a valid email format
- `@IsString()` and `@MinLength(6)` on password ensure it's a string of at least 6 characters
- `{ message: '...' }` provides custom error messages for validation failures
- We don't include `createdAt` here because clients won't send that; it's set automatically

`user.dto.ts` (Response DTO):

```
1 export class UserDto {
2   id: number;
3   email: string;
4   createdAt: Date;
5   // Notice: no password field here, since we don't want to expose it
6 }
7
```

This DTO represents what we will send to the client when returning user data, e.g., after registration or when fetching profile. It intentionally lacks the password field for security reasons.

**Hint:** class-validator is a NestJS-integrated validation tool that lets you annotate DTOs with rules. Learn more in the official docs: [Documentation | NestJS - A progressive Node.js framework](#) . Also, when you use ValidationPipe (covered soon), NestJS will automatically validate incoming requests against these DTO rules and throw an error if the input doesn't match.

## User Service [↗](#)

Let's create `user.service.ts` in the `src/user` directory:

```
1 import { Injectable, BadRequestException } from '@nestjs/common';
2 import { InjectRepository } from '@nestjs/typeorm';
3 import { Repository } from 'typeorm';
4 import { User } from '../user.entity';
5 import { CreateUserDto } from '../create-user.dto';
6 import * as bcrypt from 'bcrypt';
7
8 @Injectable()
9 export class UserService {
10   constructor(
11     @InjectRepository(User)
12     private userRepo: Repository<User>
13   ) {}
14
15   async create(createUserDto: CreateUserDto): Promise<User> {
16     const { email, password } = createUserDto;
17
18     // Check if user with the given email already exists
19     const exists = await this.userRepo.findOne({ where: { email } });
20     if (exists) {
21       throw new BadRequestException('Email already registered');
22     }
23
24     // Hash the password before saving (security best practice)
25     const saltOrRounds = 10;
26     const hashedPassword = await bcrypt.hash(password, saltOrRounds);
27
28     const newUser = this.userRepo.create({ email, password: hashedPassword });
29     return this.userRepo.save(newUser); // This will insert the new user into DB
30   }
31
32   async findByEmail(email: string): Promise<User | null> {
33     return this.userRepo.findOne({ where: { email } });
34   }
35
36   async findById(id: number): Promise<User | null> {
37     return this.userRepo.findOne({ where: { id } });
38   }
39 }
40
```

### Key points:

- We inject a TypeORM repository for the User entity using `@InjectRepository(User)`
- `create()` method handles user registration:
  - It first checks if the email is already taken and throws a `BadRequestException` if so
  - It hashes the password using `bcrypt` before storing it
  - We use `bcrypt.hash()` with a salt rounds value (10 is common)

- Never store plain-text passwords! Always hash them (bcrypt uses one-way hashing)
- `findByEmail()` and `findById()` help retrieve users for login and profile checks, respectively

## User Controller [↗](#)

Now, the controller will expose HTTP endpoints. Create `user.controller.ts` in `src/user`:

```

1  import {
2    Controller,
3    Post,
4    Body,
5    Get,
6    UseInterceptors,
7    ClassSerializerInterceptor,
8    UseGuards,
9    Req,
10 } from '@nestjs/common';
11 import { UserService } from '../user.service';
12 import { CreateUserDto } from '../create-user.dto';
13 import { User } from '../user.entity';
14 import { UserDto } from '../user.dto';
15 import { AuthGuard } from '@nestjs/passport'; // will use JWT guard for profile
16 import { instanceToPlain } from 'class-transformer';
17
18 @Controller('users')
19 @UseInterceptors(ClassSerializerInterceptor) // Apply serialization to remove excluded fields
20 export class UserController {
21   constructor(private readonly userService: UserService) {}
22
23   @Post('register')
24   async register(@Body() createUserDto: CreateUserDto): Promise<UserDto> {
25     const user = await this.userService.create(createUserDto);
26     // The saved user will have password, but thanks to ClassSerializerInterceptor and @Exclude,
27     // it will be omitted in the response
28     return instanceToPlain(user) as UserDto;
29   }
30
31   @UseGuards(AuthGuard('jwt')) // Protect this route with JWT
32   @Get('profile')
33   async getProfile(@Req() req): Promise<UserDto> {
34     // `req.user` will be set by the JwtStrategy after authentication
35     const user = await this.userService.findById(req.user.id);
36     return instanceToPlain(user) as UserDto;
37   }
38 }
39

```

### What's happening:

- We use `@Controller('users')` to prefix all routes in this controller with `/users`
- We apply `@UseInterceptors(ClassSerializerInterceptor)` at the controller level. This means any responses that are instances of classes (like our `User` entity) will be run through `class-transformer` before sending out. Because we added `@Exclude()` on `User.password`, that field will be removed
- `@Post('register')`: This is our registration route (POST `/users/register`). It expects a request body matching `CreateUserDto` (with email and password). The `ValidationPipe` (enabled globally in `main.ts`, which we'll do soon) will ensure the data meets our DTO requirements. We then call `userService.create()` to save the user. The returned `User` instance is transformed to plain object (so that password exclusion applies) and returned as `UserDto`

- `@UseGuards(AuthGuard('jwt')) @Get('profile')` : This is a protected route (GET `/users/profile`). We use NestJS's `AuthGuard('jwt')` to protect it with JWT authentication. We'll set up JWT strategy later to make this work. If the JWT is valid, `req.user` will contain the token's payload (e.g., user ID and email). We then fetch the user by ID and return it (again, using `class-transformer` to exclude the password). If the JWT is missing or invalid, the guard will automatically prevent access (returning 401 Unauthorized)

Now that our User module files are ready, we need to tell NestJS about them.

## User Module [🔗](#)

Create `user.module.ts` in `src/user`:

```
1 import { Module } from '@nestjs/common';
2 import { TypeOrmModule } from '@nestjs/typeorm';
3 import { User } from './user.entity';
4 import { UserService } from './user.service';
5 import { UserController } from './user.controller';
6
7 @Module({
8   imports: [TypeOrmModule.forFeature([User])],
9   providers: [UserService],
10  controllers: [UserController],
11  exports: [UserService], // export UserService for other modules (like AuthModule)
12 })
13 export class UserModule {}
14
```

This registers our User entity with `TypeOrmModule` (so we can use `userRepo` in the service) and sets up the service and controller.

Finally, ensure `UserModule` is imported in `AppModule` (which we did in the global TypeORM setup).

## Using class-validator for Input Validation [🔗](#)

In NestJS, validation is typically done using a Pipe. The common approach is to enable the `ValidationPipe` globally. Open `src/main.ts` and enable it:

```
1 // main.ts
2 import { ValidationPipe } from '@nestjs/common';
3
4 async function bootstrap() {
5   const app = await NestFactory.create(AppModule);
6   app.useGlobalPipes(new ValidationPipe());
7   await app.listen(3000);
8 }
9 bootstrap();
10
```

By adding `app.useGlobalPipes(new ValidationPipe())`, we tell NestJS to automatically validate all incoming requests against the DTOs we've defined using `class-validator` decorators. If a validation rule fails (e.g., email is not an email or password is too short), NestJS will respond with a 400 Bad Request and a descriptive error message. Recap of validation in our DTO:

`CreateUserDto.email` has `@IsEmail()`, so if a client sends an invalid email, the request will be rejected with an error.

`CreateUserDto.password` has `@MinLength(6)`, so "12345" would be rejected as too short. This ensures we only process valid data in our handlers, making our API more robust. Tip: The `ValidationPipe` can be customized. For example, you can strip unknown properties (to prevent extra fields in the payload) or enable auto-transformation of plain objects to class instances. See NestJS docs for more info. In our case, the defaults suffice.



## Using class-transformer to Exclude Sensitive Fields [🔗](#)

We touched on this in the User Entity and Controller. To recap: We used `@Exclude()` on `User.password`. We applied `ClassSerializerInterceptor` in the `UserController`. This combination means that any User entity returned by a controller will have the password field omitted. This is one of the main uses of class-transformer with NestJS: to serialize/transform our class instances to safe JSON. To make sure it works everywhere, some people enable the interceptor globally or use the `@UseInterceptors(ClassSerializerInterceptor)` at the controller (as we did). You can even enable it app-wide similar to `ValidationPipe`, but for simplicity, our use at the controller level is enough. Why use `@Exclude()`? From the NestJS docs: "sensitive data like passwords should always be excluded from the response". The `ClassSerializerInterceptor` uses class-transformer under the hood to achieve this. Using `@Exclude()` and returning class instances (not plain objects) ensures that "the combination of the interceptor and the entity class declaration ensures that any method that returns a User will remove the password property". Hint: We use `@Exclude()` from class-transformer to hide sensitive fields like password when returning entities. Docs: [🔗 GitHub - tystest/ack/class-transformer: Decorator-based transformation, serialization, and deserialization between objects and classes.](#) (see "Skipping specific properties") – specifically, when we transform a User instance to plain, the password will be skipped (and indeed, when NestJS sends the response, it does such a transformation). Now our user module and data handling are in place. Time to tackle authentication with Passport.js!

## Setting Up Authentication with Passport [🔗](#)

NestJS provides seamless integration with Passport, which supports many strategies. We will implement:

- Local Strategy: to validate email and password when a user logs in.
- JWT Strategy: to validate JWT tokens for protected routes (like `/users/profile`).

We'll create an Auth Module to encapsulate this logic separate from the User Module.

### Auth Module & Service [🔗](#)

Let's create `src/auth/auth.module.ts`:

```
1 import { Module } from '@nestjs/common';
2 import { PassportModule } from '@nestjs/passport';
3 import { JwtModule } from '@nestjs/jwt';
4 import { UserModule } from '../user/user.module';
5 import { AuthService } from './auth.service';
6 import { LocalStrategy } from './local.strategy';
7 import { JwtStrategy } from './jwt.strategy';
8 import { AuthController } from './auth.controller';
9
10 @Module({
11   imports: [
12     UserModule,
13     PassportModule,
14     JwtModule.register({
15       secret: 'MY_JWT_SECRET', // In a real app, use .env for secret
16       signOptions: { expiresIn: '1h' }, // Tokens expire in 1 hour
17     }),
18   ],
19   providers: [AuthService, LocalStrategy, JwtStrategy],
20   controllers: [AuthController],
21 })
22 export class AuthModule {}
23
```

Explanation: We import `UserModule` because `AuthService` will need to use `UserService` (for finding users, etc.). We import `PassportModule` (no special config needed for now). We configure `JwtModule.register` with a secret and token expiration. In practice, do not hard-code the secret; use environment variables or config service to keep it safe. For this tutorial,

"MY\_JWT\_SECRET" is a placeholder. We add AuthService, LocalStrategy, and JwtStrategy as providers so Nest can inject and use them. We add an AuthController (to handle the login route). Note: We are not using `.register({ global: true })` for JwtModule here (unlike some examples) because we'll explicitly use `AuthGuard('jwt')` for protected routes. Registering global would auto-attach JwtService everywhere, but keeping it modular is fine. Now let's create the AuthService in `src/auth/auth.service.ts`:

```
1 import { Injectable, UnauthorizedException } from '@nestjs/common';
2 import { UserService } from '../user/user.service';
3 import * as bcrypt from 'bcrypt';
4 import { JwtService } from '@nestjs/jwt';
5
6 @Injectable()
7 export class AuthService {
8   constructor(
9     private userService: UserService,
10    private jwtService: JwtService
11  ) {}
12
13  // Validate user credentials (used by local strategy)
14  async validateUser(email: string, password: string): Promise<any> {
15    const user = await this.userService.findByEmail(email);
16    if (!user) {
17      return null;
18    }
19    const passwordValid = await bcrypt.compare(password, user.password);
20    if (!passwordValid) {
21      return null;
22    }
23    // We don't want to return the password even if it's there.
24    // Using object destructuring to exclude password.
25    const { password: _, ...result } = user;
26    return result;
27  }
28
29  // Login: on successful validation, generate a JWT
30  async login(user: any) {
31    // user here is the result of validateUser (with password excluded)
32    const payload = { email: user.email, sub: user.id };
33    return { access_token: this.jwtService.sign(payload) };
34  }
35 }
36
```

What does AuthService do? `validateUser`: This is used by the local strategy. It finds the user by email, uses `bcrypt.compare` to check the provided password against the stored hashed password. If invalid, it returns null (Passport will handle this as a failed login). If valid, it returns the user data excluding the password. We use ES6 destructuring `const { password: _, ...result } = user;` to strip out the password. This follows the principle of not exposing sensitive info any further than needed. `login`: This creates a JWT token. `user` is the validated user (without password). We create a JWT payload containing the user's email and id. By convention, we use `sub` (short for subject) to hold the unique identifier of the user. This is following JWT standards for the subject claim. We then sign a token with `jwtService.sign(payload)`. The result is an object with `access_token` which we will send to the client. A Note on security: In a real application, you'd want to ensure the JWT secret is protected (via environment variables or a config service). NestJS docs emphasize not to expose the key in code. Also, consider the token expiration and whether you need refresh tokens. For simplicity, we only implement access tokens here.

## Local Strategy (Email & Password Login) [🔗](#)

Passport strategies are classes. Create `src/auth/local.strategy.ts`:

```

1 import { Strategy } from 'passport-local';
2 import { PassportStrategy } from '@nestjs/passport';
3 import { Injectable, UnauthorizedException } from '@nestjs/common';
4 import { AuthService } from '../auth.service';
5
6 @Injectable()
7 export class LocalStrategy extends PassportStrategy(Strategy) {
8   constructor(private authService: AuthService) {
9     // By default, passport-local expects 'username' and 'password'.
10    // We override to use 'email' instead of 'username'.
11    super({ usernameField: 'email' });
12  }
13
14  async validate(email: string, password: string): Promise<any> {
15    const user = await this.authService.validateUser(email, password);
16    if (!user) {
17      throw new UnauthorizedException('Invalid credentials');
18    }
19    return user;
20  }
21 }
22

```

Details: We extend `PassportStrategy(Strategy)` from `passport-local`. In `super()`, we pass `{ usernameField: 'email' }` to tell Passport to treat the 'email' field as the username. This way we can send `{ email, password }` in our login requests. The `validate()` method is automatically called by Passport when `AuthGuard('local')` is used (we'll use it in the controller). It calls `AuthService.validateUser`. If no user or password is wrong, it throws `UnauthorizedException`. If ok, it returns the user (without password, as set in `AuthService`). Returning a value means the user is authenticated and that returned value will be attached to `req.user`. So the local strategy handles verifying email/password on login requests.

## JWT Strategy (Securing Protected Routes) [🔗](#)

Next, create `src/auth/jwt.strategy.ts`:

```

1 import { Injectable } from '@nestjs/common';
2 import { PassportStrategy } from '@nestjs/passport';
3 import { ExtractJwt, Strategy } from 'passport-jwt';
4
5 @Injectable()
6 export class JwtStrategy extends PassportStrategy(Strategy) {
7   constructor() {
8     super({
9       // Extract JWT from the Authorization header as Bearer token
10      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
11      ignoreExpiration: false, // JWT expiration handled by Passport, false means reject expired tokens
12      secretOrKey: 'MY_JWT_SECRET', // same secret as used in JwtModule
13    });
14  }
15
16  async validate(payload: any) {
17    // Passport automatically verifies JWT and decodes payload
18    // Here, we simply return the payload as the user object (could fetch user from DB as well)
19    return { id: payload.sub, email: payload.email };
20  }
21 }
22

```

Explanation: We configure passport-jwt to extract the token from the Bearer token in the auth header. The standard is Authorization: Bearer . ignoreExpiration: false means if the token is expired, it will be rejected. secretOrKey must match the secret we used to sign tokens (again, better to use a config service to inject this value for maintainability). The validate(payload: any) function: Is called after the JWT is verified. payload is whatever we put in the token (for us, { sub: [user.id](#), email: user.email }). We return an object representing the "user" for the request. Here, we simply return { id: payload.sub, email: payload.email }. This object becomes req.user in any route guarded by JWT (like our profile route). We could also optionally re-fetch the user from the database using payload.sub (userId) to ensure the user still exists or to get fresh roles/permissions. But for our case, the token itself is enough. The strategies (LocalStrategy and JwtStrategy) will be automatically picked up by Nest because we provided them in AuthModule's providers.

## Auth Controller (Login & JWT issuance)

Finally, let's create a controller for authentication in src/auth/auth.controller.ts:

```
1 import { Controller, Post, UseGuards, Request } from '@nestjs/common';
2 import { AuthService } from '../auth.service';
3 import { AuthGuard } from '@nestjs/passport';
4
5 @Controller('auth')
6 export class AuthController {
7   constructor(private authService: AuthService) {}
8
9   @UseGuards(AuthGuard('local')) // Uses the local strategy to validate user
10  @Post('login')
11  async login(@Request() req) {
12    // If we're here, the user is validated (req.user is set by LocalStrategy)
13    return this.authService.login(req.user);
14  }
15 }
16
```

What's happening: @Controller('auth'): Prefixes routes with /auth. @Post('login'): This will handle POST /auth/login.

@UseGuards(AuthGuard('local')): We apply the local auth guard. When a request hits this route, the guard will trigger the LocalStrategy's validate method with the credentials provided. If authentication fails, a 401 is returned and the function body (login()) won't execute. If it succeeds, req.user will contain the user object returned by LocalStrategy. Inside login(), we simply return this.authService.login(req.user). AuthService.login() will create a JWT and return it. The response sent to the client will be something like: { "access\_token": "<JWT\_TOKEN>" }. At this point, our authentication system is ready: /auth/login for logging in (with email & password). /users/register for creating a new account. /users/profile for getting the current user profile (protected by JWT). Don't forget to import AuthModule into AppModule (so open app.module.ts and add AuthModule to the imports array). Your AppModule imports might look like:

```
1 import { Module } from '@nestjs/common';
2 import { TypeOrmModule } from '@nestjs/typeorm';
3 import { UserModule } from '../user/user.module';
4 import { AuthModule } from '../auth/auth.module';
5
6 @Module({
7   imports: [
8     TypeOrmModule.forRoot({ ... }),
9     UserModule,
10    AuthModule,
11  ],
12 })
13 export class AppModule {}
14
```

Now, let's recap and test the back-end (if you run it) before moving to the front-end:

Registration: POST /users/register with JSON body {"email": "...", "password": "..."} should create a new user and return their data (id, email, createdAt). Password is excluded from the response thanks to @Exclude.

Login: POST /auth/login with JSON body {"email": "...", "password": "..."} (for a user that exists) should return an access\_token (JWT string).

Profile: GET /users/profile with header Authorization: Bearer (the token from login) should return the user's data (id, email, createdAt). If no token or invalid token, it should return 401 Unauthorized.

Everything in the back-end should now be set up. 🎉 Now, onto the front-end to interact with these endpoints.

## Implementing Registration & Profile Routes [↗](#)

(This section was covered in the controller, but let's ensure the tutorial flow remains logical.)

We already implemented the core routes in the UserController: /users/register (POST): Public route for registering a new user. Calls UserService.create to add a user. Returns the created user info (excluding password). /users/profile (GET): Protected route, requires JWT. Returns current user info. And in the AuthController: /auth/login (POST): Public route for logging in. On success, returns a JWT token.

Let's ensure we've tested these via something like cURL or Postman:

```
1 # Register a user
2 curl -X POST http://localhost:3000/api/users/register \
3 -H 'Content-Type: application/json' \
4 -d '{"email": "test@example.com", "password": "strongpass"}'
5
```

Expected Response (201 Created):

```
1 { "id": 1, "email": "test@example.com", "createdAt": "2025-03-31T...Z" }
2
```

```
1 # Login with that user
2 curl -X POST http://localhost:3000/auth/login \
3 -H 'Content-Type: application/json' \
4 -d '{"email": "test@example.com", "password": "strongpass"}'
5
```

Expected Response (200 OK):

```
1 { "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6Ii..." }
2
```

```
1 # Access profile with token (replace <TOKEN> with the actual token string)
2 curl http://localhost:3000/users/profile \
3 -H 'Authorization: Bearer <TOKEN>'
4
```

Expected Response (200 OK):

```
1 { "id": 1, "email": "test@example.com", "createdAt": "2025-03-31T...Z" }
2
```

If all good, our backend is working! Now, let's build a simple React front-end to consume this API.

## Building the React Frontend [↗](#)

Now we'll build a simple React application with the following parts:

1. A Registration form (/register page)
2. A Login form (/login page)
3. A Profile page (/profile page) that shows user info after logging in.

For simplicity, we can make this a single-page app with conditional rendering, since focusing on React Router isn't the main goal here. Alternatively, one can use React Router to handle multi-page flow, but we'll keep it straightforward.

## Project Setup for React [🔗](#)

Initialize a new React app (we can use Create React App or Vite for simplicity). Let's use Create React App here:

```
1 npx create-react-app user-auth-client
2 cd user-auth-client
3
```

Now, install axios for making HTTP requests (or you can use fetch API directly):

```
1 npm install axios
2
```

And start the development server:

```
1 npm start
2
```

This should open a React app on <http://localhost:3000/> (if port 3000 is not free, it might use 3001). We'll edit the src/App.js (assuming a JavaScript React app). For clarity, let's use a minimal approach and avoid too many extra components. We'll manage a simple state for which "view" we're on (register, login, profile) and store the token + user info. Plan: Use React's useState to keep track of: view ('register' | 'login' | 'profile' to know which form or page to show). token (JWT string after login). user (user info like email and id after registration or profile fetch). Implement handlers for register and login that call our API, then update state accordingly. After login, automatically fetch the profile (or rely on the login response plus perhaps get profile again). This is a simple approach. In a more complex app, you'd use context or Redux for auth state, but we'll keep it simple.

## Registration Form [🔗](#)

In App.js (or separate component if you prefer):

```
1 import React, { useState } from 'react';
2 import axios from 'axios';
3
4 function App() {
5   const [view, setView] = useState('register'); // default to registration view for now
6   const [token, setToken] = useState(null);
7   const [user, setUser] = useState(null);
8
9   // Form state
10  const [email, setEmail] = useState('');
11  const [password, setPassword] = useState('');
12
13  const handleRegister = async () => {
14    try {
15      const response = await axios.post('http://localhost:3000/users/register', {
16        email,
17        password,
18      });
19      alert('Registration successful! You can now log in.');// Optionally, auto-login or auto-switch to
      login view
20      setView('login');
```

```

21     setEmail('');
22     setPassword('');
23   } catch (error) {
24     console.error(error);
25     if (error.response) {
26       alert('Registration failed: ' + error.response.data.message);
27     } else {
28       alert('Registration failed: Network or server error');
29     }
30   }
31 };
32
33 const handleLogin = async () => {
34   try {
35     const response = await axios.post('http://localhost:3000/auth/login', { email, password });
36     const { access_token } = response.data;
37     setToken(access_token); // Once logged in, fetch profile
38     const profileRes = await axios.get('http://localhost:3000/users/profile', {
39       headers: { Authorization: `Bearer ${access_token}` },
40     });
41     setUser(profileRes.data);
42     setView('profile');
43   } catch (error) {
44     console.error(error);
45     if (error.response) {
46       alert('Login failed: ' + error.response.data.message);
47     } else {
48       alert('Login failed: Network or server error');
49     }
50   }
51 };
52
53 const logout = () => {
54   setToken(null);
55   setUser(null);
56   setEmail('');
57   setPassword('');
58   setView('login');
59 };
60
61 // Render different views based on state
62 if (view === 'register') {
63   return (
64     <div className="App">
65       <h2>Register</h2>
66       <input
67         type="email"
68         placeholder="Email"
69         value={email}
70         onChange={e => setEmail(e.target.value)}
71       />
72       <br />
73       <input
74         type="password"
75         placeholder="Password"
76         value={password}
77         onChange={e => setPassword(e.target.value)}
78       />

```

```

79     <br />
80     <button onClick={handleRegister}>Register</button>
81     <p>
82         {' '}
83         Already have an account? <button onClick={() => setView('login')}>Login</button>{' '}
84     </p>
85 </div>
86 );
87 }
88
89 if (view === 'login') {
90     return (
91         <div className="App">
92             <h2>Login</h2>
93             <input
94                 type="email"
95                 placeholder="Email"
96                 value={email}
97                 onChange={e => setEmail(e.target.value)}
98             />
99             <br />
100             <input
101                 type="password"
102                 placeholder="Password"
103                 value={password}
104                 onChange={e => setPassword(e.target.value)}
105             />
106             <br />
107             <button onClick={handleLogin}>Login</button>
108             <p>
109                 {' '}
110                 No account? <button onClick={() => setView('register')}>Register</button>{' '}
111             </p>
112         </div>
113     );
114 }
115
116 if (view === 'profile') {
117     return (
118         <div className="App">
119             <h2>User Profile</h2>
120             {user ? (
121                 <div>
122                     <p>
123                         <strong>ID:</strong> {user.id}
124                     </p>
125                     <p>
126                         <strong>Email:</strong> {user.email}
127                     </p>
128                     <p>
129                         <strong>Joined:</strong> {new Date(user.createdAt).toLocaleString()}
130                     </p>
131                 </div>
132             ) : (
133                 <p>Loading profile...</p>
134             )}
135             <button onClick={logout}>Logout</button>
136         </div>

```



```

137     );
138   }
139
140   return null; // should not reach here
141 }
142
143 export default App;
144

```

Let's break down this React code: We manage view state to toggle between "register", "login", and "profile" interfaces. We bind inputs for email and password to state. `handleRegister`: Calls POST `/users/register` on our NestJS API with the email and password. On success, we alert the user and switch to the login view. We clear the form fields. `handleLogin`: Calls POST `/auth/login`. On success, we get `access_token`. We store it in state, then immediately make a request to `/users/profile` with the token in the Authorization header. Upon success, we set the user state with profile data and switch to the profile view. We also define a logout function to clear everything and go back to login. Using the JWT token on the frontend: We stored it in the token state and also kept it around to fetch profile. In a more persistent scenario, you might want to store the token in `localStorage` or a cookie so it remains after a page refresh. For example, on successful login, you could do:

```

1 localStorage.setItem('token', access_token);
2

```

and on app start, check `localStorage` to set the token if it exists (and adjust the view accordingly). But careful: storing JWT in `localStorage` is common but has some XSS risk if your site can be attacked by script injection. A more secure approach is to store tokens in `HttpOnly` cookies. For simplicity, `localStorage` or memory is fine for now. Tip: When sending the JWT from React for protected routes, always include it in the Authorization header as a Bearer token. Example: Authorization: Bearer

## Login Form [↗](#)

The login form was included above in the same App component for simplicity. It allows entering email and password, and calls `handleLogin` when clicking the login button.

## Profile Page (Fetching Protected Data) [↗](#)

Our profile view simply displays the user state which we fetched right after login. The profile is fetched by calling our API with the JWT, showing that part of the flow. If we refresh the page in this simplistic app, we'll lose the state and go back to default (register view). As mentioned, implementing persistence (like checking `localStorage` for a token and then fetching profile) is a good enhancement.

## Storing and Using the JWT Token [↗](#)

For clarity: After login, we have `access_token`. In this example, we kept it in a React state and used it immediately to fetch profile. If this were a multi-page app, you'd typically save the token (e.g., in `localStorage` or a global context) and include it in future requests. Every time we need to access a protected route (like profile, or maybe updating user settings, etc.), we must include the Authorization header. Axios allows setting a default header or you can attach per request, as shown. Hint: Always protect the JWT. Do not expose it in URLs or anywhere it could be leaked. Using an HTTP-only cookie is more secure against XSS, but reading it in React (for Authorization header) requires either a proxy or cookie + server reads it. For learning purposes, using `localStorage` is okay, just be mindful of XSS. Also, never store sensitive info inside the JWT payload without encryption — while JWTs are signed (to prevent tampering), their contents are readable by anyone having the token (it's just base64 encoded).

## Conclusion & Next Steps [↗](#)

Congratulations! You've built a basic full-stack user management system with:

- NestJS backend (with a modular structure using a `UserModule` and `AuthModule`).
- PostgreSQL as the database and TypeORM for interacting with it via the User entity.
- Passport with local and JWT strategies for authentication.

- Users can register, log in, and access protected routes with a JWT.
- class-validator to ensure incoming data is well-formed (preventing things like empty emails or short passwords).
- class-transformer (with @Exclude) to make sure we never send passwords in responses.
- ReactJS frontend with simple forms to allow users to register and log in, and a profile page that shows data only when authenticated.

Throughout this tutorial, we followed a real implementation flow: set up the environment, built the NestJS backend step by step, then created a React frontend to use the API, mirroring how you'd develop a real project. Where to go from here? Some ideas:

- Add better error handling and feedback on the frontend.
- Implement refresh tokens or a logout mechanism on the backend (our logout on frontend just dumps the token).
- Enhance security: use HTTPS, secure cookies for JWT, password reset flow, etc.
- Add more fields to the User (like name, roles) and implement role-based authorization for certain routes.
- Explore NestJS Guards and Interceptors further to add features like rate limiting, logging, etc., to your API.
- Write tests for your NestJS services and controllers.

Helpful Resources:

- NestJS Official Docs on Authentication – covers JWT strategy as we did and more.
- NestJS Docs on Validation – for understanding ValidationPipe and class-validator usage.
- NestJS Docs on Serialization – for using class-transformer (@Exclude, @Expose).
- TypeORM Repository API for advanced database operations.
- React Docs or tutorials for managing global auth state if you want to scale the frontend.

Happy coding! Enjoy building secure and scalable full-stack apps with NestJS and React.