CRUD generator

Throughout the life span of a project, when we build new features, we often need to add new resources to our application. These resources typically require multiple, repetitive operations that we have to repeat each time we define a new resource.

Introduction

Let's imagine a real-world scenario, where we need to expose CRUD endpoints for 2 entities, let's say **User** and **Product** entities. Following the best practices, for each entity we would have to perform several operations, as follows:

- Generate a module (`nest g mo`) to keep code organized and establish clear boundaries (grouping related components)
- Generate a controller (`nest g co`) to define CRUD routes (or queries/mutations for GraphQL applications)
- Generate a service (`nest g s`) to implement & isolate business logic
- Generate an entity class/interface to represent the resource data shape
- Generate Data Transfer Objects (or inputs for GraphQL applications) to define how the data will be sent over the network

That's a lot of steps!

To help speed up this repetitive process, Nest CLI provides a generator (schematic) that automatically generates all the boilerplate code to help us avoid doing all of this, and make the developer experience much simpler.

> info **Note** The schematic supports generating **HTTP** controllers, **Microservice** controllers, **GraphQL** resolvers (both code first and schema first), and **WebSocket** Gateways.

Generating a new resource

To create a new resource, simply run the following command in the root directory of your project:

```
$ nest g resource
```

`nest g resource` command not only generates all the NestJS building blocks (module, service, controller classes) but also an entity class, DTO classes as well as the testing (`.spec`) files.

Below you can see the generated controller file (for REST API):

```
@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);
  }

  @Patch(':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);
  }
}
```

Also, it automatically creates placeholders for all the CRUD endpoints (routes for REST APIs, queries and mutations for GraphQL, message subscribes for both Microservices and WebSocket Gateways) - all without having to lift a finger.

> warning **Note** Generated service classes are **not** tied to any specific **ORM (or data source)**. This makes the generator generic enough to meet the needs of any project. By default, all methods will contain placeholders, allowing you to populate it with the data sources specific to your project.

Likewise, if you want to generate resolvers for a GraphQL application, simply select the `GraphQL (code first)` (or `GraphQL (schema first)`) as your transport layer.

In this case, NestJS will generate a resolver class instead of a REST API controller:

```
$ nest g resource users

> ? What transport layer do you use? GraphQL (code first)
> ? Would you like to generate CRUD entry points? Yes
> CREATE src/users/users.module.ts (224 bytes)
> CREATE src/users/users.resolver.spec.ts (525 bytes)
> CREATE src/users/users.resolver.ts (1109 bytes)
> CREATE src/users/users.service.spec.ts (453 bytes)
> CREATE src/users/users.service.ts (625 bytes)
> CREATE src/users/dto/create-user.input.ts (195 bytes)
> CREATE src/users/dto/update-user.input.ts (281 bytes)
> CREATE src/users/entities/user.entity.ts (187 bytes)
> UPDATE src/app.module.ts (312 bytes)
```

> info **Hint** To avoid generating test files, you can pass the `--no-spec` flag, as follows: `nest g resource users --no-spec`

We can see below, that not only were all boilerplate mutations and queries created, but everything is all tied together. We're utilizing the `UsersService`, `User` Entity, and our DTO's.

```
import { Resolver, Query, Mutation, Args, Int } from '@nestjs/graphql';
import { UsersService } from './users.service';
import { User } from './entities/user.entity';
import { CreateUserInput } from './dto/create-user.input';
import { UpdateUserInput } from './dto/update-user.input';

@Resolver(() => User)
export class UsersResolver {
  constructor(private readonly usersService: UsersService) {}

  @Mutation(() => User)
  createUser(@Args('createUserInput') createUserInput: CreateUserInput) {
    return this.usersService.create(createUserInput);
  }

  @Query(() => [User], { name: 'users' })
  findAll() {
    return this.usersService.findAll();
  }

  @Query(() => User, { name: 'user' })
  findOne(@Args('id', { type: () => Int }) id: number) {
    return this.usersService.findOne(id);
  }

  @Mutation(() => User)
  updateUser(@Args('updateUserInput') updateUserInput: UpdateUserInput) {
    return this.usersService.update(updateUserInput.id, updateUserInput);
  }

  @Mutation(() => User)
  removeUser(@Args('id', { type: () => Int }) id: number) {
    return this.usersService.remove(id);
  }
}
```