CRUD

This chapter applies only to TypeScript

CRUD package (`@nestjsx/crud`) helps you create CRUD controllers and services with ease and provides a bunch of the features for your RESTful API out of the box:

- Database agnostic extendable CRUD controller
- Query string parsing with filtering, pagination, sorting, relations, nested relations, cache, etc.
- Framework agnostic package with query builder for frontend usage
- Query, path params and DTO validation
- Overriding controller methods with ease
- Tiny but powerful configuration (including global configuration)
- Additional helper decorators
- Swagger documentation

> **NOTICE**
> So far, `@nestjsx/crud` only supports `TypeORM` , but other ORMs like `Sequelize` and `Mongoose` will be included in the near future. So in this article, you'll learn how to create CRUD controllers and services using `TypeORM` . We assume that you have already successfully installed and set up the `@nestjs/typeorm` package. To learn more, see here.

Getting started

To start creating CRUD functionality we have to install all required dependencies:

```
npm i --save @nestjsx/crud @nestjsx/crud-typeorm class-transformer class-validator
```

Assuming that you already have some **entities** in your project:

hero.entity.ts JS

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

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

```
  @Column()
  name: string;

  @Column({ type: 'number' })
  power: number;
}
```

The first step we need to do is to create a **service**:

heroes.service.ts JS

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { TypeOrmCrudService } from '@nestjsx/crud-typeorm';
import { Hero } from './hero.entity';

@Injectable()
export class HeroesService extends TypeOrmCrudService<Hero> {
  constructor(@InjectRepository(Hero) repo) {
    super(repo);
  }
}
```

We're done with the service so let's create a **controller**:

heroes.controller.ts JS

```
import { Controller } from '@nestjs/common';
import { Crud } from '@nestjsx/crud';
import { Hero } from './hero.entity';
import { HeroesService } from './heroes.service';

@Crud({
  model: {
    type: Hero,
  },
})
@Controller('heroes')
export class HeroesController {
  constructor(public service: HeroesService) {}
}
```

And finally, we need to wire up everything in our **module**:

heroes.module.ts JS

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';

import { Hero } from './hero.entity';
import { HeroesService } from './heroes.service';
import { HeroesController } from './heroes.controller';

@Module({
  imports: [TypeOrmModule.forFeature([Hero])],
  providers: [HeroesService],
  controllers: [HeroesController],
})
export class HeroesModule {}
```

> **NOTICE**
> Do not forget to import the `HeroesModule` into the root `ApplicationModule`.

Afterwards, your Nest application will have these newly created endpoints:

- `GET /heroes` - get many heroes.
- `GET /heroes/:id` - get one hero.
- `POST /heroes/bulk` - create many heroes.
- `POST /heroes` - create one hero.
- `PATCH /heroes/:id` - update one hero.
- `PUT /heroes/:id` - replace one hero.
- `DELETE /heroes/:id` - delete one hero.

Filtering and pagination

CRUD provides rich tools for filtering and pagination. Example request:

> **REQUEST**
> GET /heroes?**select**=name&**filter**=power||gt||90&**sort**=name,ASC&**page**=1&**limit**=3

In this example, we requested the list of heroes and selected `name` attribute only, where the `power` of a hero is greater than 90, and set a result limit to 3 within page 1, and sorted by `name` in `ASC` order.

The response object will be similar to this one:

```
{
  "data": [
    {
      "id": 2,
      "name": "Batman"
    },
    {
      "id": 4,
      "name": "Flash"
    },
    {
      "id": 3,
      "name": "Superman"
    }
  ],
  "count": 3,
  "total": 14,
  "page": 1,
  "pageCount": 5
}
```

> **NOTICE**
> Primary columns persist in the resource response object whether they were requested or not. In our case, it's an `id` column.

The complete list of query params and filter operators can be found in the project's **Wiki**.

Relations

Another feature that is worth mentioning is "relations". In your CRUD controller, you can specify the list of entity's relations which are allowed to fetch within your API calls:

```
@Crud({
  model: {
    type: Hero,
  },
  join: {
    profile: {
      exclude: ['secret'],
    }
```

```
    faction: {
      eager: true,
      only: ['name'],
    },
  },
})
@Controller('heroes')
export class HeroesController {
  constructor(public service: HeroesService) {}
}
```

After specifying allowed relations in the `@Crud()` decorator options, you can make the following request:

> **REQUEST**
> GET /heroes/25?**join**=profile||address,bio

The response will contain a hero object with a joined profile which will have `address` and `bio` columns selected.

Also, the response will contain a `faction` object with the `name` column selected because it was set to `eager: true` and thus persists in every response.

You can find more information about relations in the project's **WiKi**.

Path params validation

By default, **CRUD** will create a slug with the name `id` and will be validating it as a `number`.

But there is a possibility to change this behavior. Assume, your entity has a primary column `_id` - a UUID string - and you need to use it as a slug for your endpoints. With these options it's easy to do:

```
@Crud({
  model: {
    type: Hero,
  },
  params: {
    _id: {
      field: '_id',
      type: 'uuid',
      primary: true,
    },
  },
})
@Controller('heroes')
export class HeroesController {
```

```
export class HeroesController {
  constructor(public service: HeroesService) {}
}
```

For more params options please see the project's **Wiki**.

Request body validation

Request body validation is performed out of the box by applying Nest `ValidationPipe` on each POST, PUT, PATCH request. We use `model.type` from `@Crud()` decorator options as a DTO that describes validation rules.

In order to do that properly we use **validation groups**:

hero.entity.ts JS

```
import { Entity, PrimaryGeneratedColumn, Column } from 'typeorm';
import { IsOptional, IsDefined, IsString, IsNumber } from 'class-validator';
import { CrudValidationGroups } from '@nestjsx/crud';

const { CREATE, UPDATE } = CrudValidationGroups;

@Entity()
export class Hero {
  @IsOptional({ always: true })
  @PrimaryGeneratedColumn()
  id: number;

  @IsOptional({ groups: [UPDATE] })
  @IsDefined({ groups: [CREATE] })
  @IsString({ always: true })
  @Column()
  name: string;

  @IsOptional({ groups: [UPDATE] })
  @IsDefined({ groups: [CREATE] })
  @IsNumber({}, { always: true })
  @Column({ type: 'number' })
  power: number;
}
```

> **NOTICE**
> The full support of separate DTO classes for `create` and `update` actions is one of the main priorities for the next **CRUD** release.

Routes options

You can disable or enable only some particular routes that are being generated by applying `@Crud()` decorator:

```
@Crud({
  model: {
    type: Hero,
  },
  routes: {
    only: ['getManyBase'],
    getManyBase: {
      decorators: [
        UseGuards(HeroAuthGuard)
      ]
    }
  }
})
@Controller('heroes')
export class HeroesController {
  constructor(public service: HeroesService) {}
}
```

Also, you can apply any method decorators by passing them to the specific route `decorators` array. This is convenient when you want to add some decorators without overriding base methods.

Documentation

The examples in this chapter cover only some of the **CRUD** features. You can find answers to many more usage questions on the project's **Wiki** page.

Support us

Nest is an MIT-licensed open source project. It can grow thanks to the support by these awesome people. If you'd like to join them, please read more **here**.

Principal Sponsor

Sponsors / Partners

Become a sponsor

SOFTWARE