Mongo

There are two ways of dealing with the MongoDB database. You can either use an **ORM** that provides a MongoDB support or **Mongoose** which is the most popular **MongoDB** object modeling tool. If you wanna stay with the **ORM** you can follow these steps. Otherwise, we'll use the dedicated `@nestjs/mongoose` package.

Firstly, we need to install all of the required dependencies:

```
$ npm install --save @nestjs/mongoose mongoose
```

Once the installation process is completed, we can import the `MongooseModule` into the root `ApplicationModule`.

app.module.ts JS

```
import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';

@Module({
  imports: [MongooseModule.forRoot('mongodb://localhost/nest')],
})
export class ApplicationModule {}
```

The `forRoot()` method accepts the same configuration object as `mongoose.connect()` from the **Mongoose** package.

Model injection

With Mongoose, everything is derived from a **Schema**. Let's define the `CatSchema`:

schemas/cat.schema.ts JS

```
import * as mongoose from 'mongoose';

export const CatSchema = new mongoose.Schema({
  name: String,
  age: Number,
  breed: String,
});
```

The `CatsSchema` belongs to the `cats` directory. This directory represents the `CatsModule`. It's your decision where you gonna keep your schema files. From our point of view, the best way's to hold them nearly their **domain**, in the appropriate module directory.

Let's have a look at the `CatsModule`:

cats.module.ts JS

```
import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';
import { CatSchema } from './schemas/cat.schema';

@Module({
  imports: [MongooseModule.forFeature([{ name: 'Cat', schema: CatSchema }])],
  controllers: [CatsController],
  providers: [CatsService],
})
export class CatsModule {}
```

This module uses `forFeature()` method to define which models shall be registered in the current scope. Thanks to that, we can inject the `CatModel` to the `CatsService` using the `@InjectModel()` decorator:

cats.service.ts JS

```
import { Model } from 'mongoose';
import { Injectable } from '@nestjs/common';
import { InjectModel } from '@nestjs/mongoose';
import { Cat } from './interfaces/cat.interface';
import { CreateCatDto } from './dto/create-cat.dto';

@Injectable()
export class CatsService {
  constructor(@InjectModel('Cat') private readonly catModel: Model<Cat>) {}

  async create(createCatDto: CreateCatDto): Promise<Cat> {
    const createdCat = new this.catModel(createCatDto);
    return await createdCat.save();
  }

  async findAll(): Promise<Cat[]> {
    return await this.catModel.find().exec();
  }
```

```
  }
}
```

Testing

When it comes to unit test our application, we usually want to avoid any database connection, making our test suits independent and their execution process quick as possible. But our classes might depend on models that are pulled from the connection instance. What's then? The solution is to create fake models. In order to achieve that, we should set up **custom providers** . In fact, each registered model is represented by `NameModel` token, where `Name` is a model's name.

The `@nestjs/mongoose` package exposes `getModelToken()` function that returns prepared token based on a given model's name.

```
@Module({
  providers: [
    CatsService,
    {
      provide: getModelToken('Cat'),
      useValue: catModel,
    },
  ],
})
export class CatsModule {}
```

Now a hardcoded `catModel` will be used as a `Model<Cat>` . Whenever any provider asks for `Model<Cat>` using an `@InjectModel()` decorator, Nest will use a registered `catModel` object.

Async configuration

Quite often you might want to asynchronously pass your module options instead of passing them beforehand. In such case, use `forRootAsync()` method, that provides a couple of various ways to deal with async data.

First possible approach is to use a factory function:

```
MongooseModule.forRootAsync({
  useFactory: () => ({
    uri: 'mongodb://localhost/nest',
  }),
});
```

Obviously, our factory behaves like every other one (might be `async` and is able to inject dependencies through

`inject`).

```
MongooseModule.forRootAsync({
  imports: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
    uri: configService.getString('MONGODB_URI'),
  }),
  inject: [ConfigService],
});
```

Alternatively, you are able to use class instead of a factory.

```
MongooseModule.forRootAsync({
  useClass: MongooseConfigService,
});
```

Above construction will instantiate `MongooseConfigService` inside `MongooseModule` and will leverage it to create options object. The `MongooseConfigService` has to implement `MongooseOptionsFactory` interface.

```
@Injectable()
class MongooseConfigService implements MongooseOptionsFactory {
  createMongooseOptions(): MongooseModuleOptions {
    return {
      uri: 'mongodb://localhost/nest',
    };
  }
}
```

In order to prevent the creation of `MongooseConfigService` inside `MongooseModule` and use a provider imported from a different module, you can use the `useExisting` syntax.

```
MongooseModule.forRootAsync({
  imports: [ConfigModule],
  useExisting: ConfigService,
});
```

It works the same as `useClass` with one critical difference - `MongooseModule` will lookup imported modules to reuse

already created `ConfigService`, instead of instantiating it on its own.

Example

A working example is available here.

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