MongoDB (Mongoose)

> **Warning** In this article, you'll learn how to create a `DatabaseModule` based on the **Mongoose** package from scratch using custom components. As a consequence, this solution contains a lot of overhead that you can omit using ready to use and available out-of-the-box dedicated `@nestjs/mongoose` package. To learn more, see here.

Mongoose is the most popular MongoDB object modeling tool.

Getting started

To start the adventure with this library we have to install all required dependencies:

```
$ npm install --save mongoose
```

The first step we need to do is to establish the connection with our database using `connect()` function. The `connect()` function returns a `Promise`, and therefore we have to create an async provider.

```
@@filename(database.providers)
import * as mongoose from 'mongoose';

export const databaseProviders = [
  {
    provide: 'DATABASE_CONNECTION',
    useFactory: (): Promise<typeof mongoose> =>
      mongoose.connect('mongodb://localhost/nest'),
  },
];
@@switch
import * as mongoose from 'mongoose';

export const databaseProviders = [
  {
    provide: 'DATABASE_CONNECTION',
    useFactory: () => mongoose.connect('mongodb://localhost/nest'),
  },
];
```

> info **Hint** Following best practices, we declared the custom provider in the separated file which has a `*.providers.ts` suffix.

Then, we need to export these providers to make them **accessible** for the rest part of the application.

```
@@filename(database.module)
import { Module } from '@nestjs/common';
import { databaseProviders } from './database.providers';
```

```
@Module({
  providers: [...databaseProviders],
  exports: [...databaseProviders],
})
export class DatabaseModule {}
```

Now we can inject the `Connection` object using `@Inject()` decorator. Each class that would depend on the `Connection` async provider will wait until a `Promise` is resolved.

Model injection

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

```
@@filename(schemas/cat.schema)
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`.

Now it's time to create a **Model** provider:

```
@@filename(cats.providers)
import { Connection } from 'mongoose';
import { CatSchema } from './schemas/cat.schema';

export const catsProviders = [
  {
    provide: 'CAT_MODEL',
    useFactory: (connection: Connection) => connection.model('Cat',
CatSchema),
    inject: ['DATABASE_CONNECTION'],
  },
];
@@switch
import { CatSchema } from './schemas/cat.schema';

export const catsProviders = [
  {
    provide: 'CAT_MODEL',
    useFactory: (connection) => connection.model('Cat', CatSchema),
    inject: ['DATABASE_CONNECTION'],
  },
];
```

> warning **Warning** In the real-world applications you should avoid **magic strings**. Both `CAT_MODEL` and `DATABASE_CONNECTION` should be kept in the separated `constants.ts` file.

Now we can inject the `CAT_MODEL` to the `CatsService` using the `@Inject()` decorator:

```
@@filename(cats.service)
import { Model } from 'mongoose';
import { Injectable, Inject } from '@nestjs/common';
import { Cat } from './interfaces/cat.interface';
import { CreateCatDto } from './dto/create-cat.dto';

@Injectable()
export class CatsService {
  constructor(
    @Inject('CAT_MODEL')
    private catModel: Model<Cat>,
  ) {}

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

  async findAll(): Promise<Cat[]> {
    return this.catModel.find().exec();
  }
}
@@switch
import { Injectable, Dependencies } from '@nestjs/common';

@Injectable()
@Dependencies('CAT_MODEL')
export class CatsService {
  constructor(catModel) {
    this.catModel = catModel;
  }

  async create(createCatDto) {
    const createdCat = new this.catModel(createCatDto);
    return createdCat.save();
  }

  async findAll() {
    return this.catModel.find().exec();
  }
}
```

In the above example we have used the `Cat` interface. This interface extends the `Document` from the mongoose package:

```
import { Document } from 'mongoose';

export interface Cat extends Document {
  readonly name: string;
  readonly age: number;
  readonly breed: string;
}
```

The database connection is **asynchronous**, but Nest makes this process completely invisible for the end-user. The `CatModel` class is waiting for the db connection, and the `CatsService` is delayed until model is ready to use. The entire application can start when each class is instantiated.

Here is a final `CatsModule`:

```
@@filename(cats.module)
import { Module } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';
import { catsProviders } from './cats.providers';
import { DatabaseModule } from '../database/database.module';

@Module({
  imports: [DatabaseModule],
  controllers: [CatsController],
  providers: [
    CatsService,
    ...catsProviders,
  ],
})
export class CatsModule {}
```

> info **Hint** Do not forget to import the `CatsModule` into the root `AppModule`.

Example

A working example is available here.