SQL (Sequelize)

This chapter applies only to TypeScript

> **Warning** In this article, you'll learn how to create a `DatabaseModule` based on the **Sequelize** package from scratch using custom components. As a consequence, this technique contains a lot of overhead that you can avoid by using the dedicated, out-of-the-box `@nestjs/sequelize` package. To learn more, see here.

Sequelize is a popular Object Relational Mapper (ORM) written in a vanilla JavaScript, but there is a sequelize-typescript TypeScript wrapper which provides a set of decorators and other extras for the base sequelize.

Getting started

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

```
$ npm install --save sequelize sequelize-typescript mysql2
$ npm install --save-dev @types/sequelize
```

The first step we need to do is create a **Sequelize** instance with an options object passed into the constructor. Also, we need to add all models (the alternative is to use `modelPaths` property) and `sync()` our database tables.

```
@@filename(database.providers)
import { Sequelize } from 'sequelize-typescript';
import { Cat } from '../cats/cat.entity';

export const databaseProviders = [
  {
    provide: 'SEQUELIZE',
    useFactory: async () => {
      const sequelize = new Sequelize({
        dialect: 'mysql',
        host: 'localhost',
        port: 3306,
        username: 'root',
        password: 'password',
        database: 'nest',
      });
      sequelize.addModels([Cat]);
      await sequelize.sync();
      return sequelize;
    },
  },
];
```

> 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.

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

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

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

Model injection

In Sequelize the **Model** defines a table in the database. Instances of this class represent a database row. Firstly, we need at least one entity:

```
@@filename(cat.entity)
import { Table, Column, Model } from 'sequelize-typescript';

@Table
export class Cat extends Model {
  @Column
  name: string;

  @Column
  age: number;

  @Column
  breed: string;
}
```

The `Cat` entity belongs to the `cats` directory. This directory represents the `CatsModule`. Now it's time to create a **Repository** provider:

```
@@filename(cats.providers)
import { Cat } from './cat.entity';

export const catsProviders = [
  {
    provide: 'CATS_REPOSITORY',
    useValue: Cat,
```

```
  },
];
```

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

In Sequelize, we use static methods to manipulate the data, and thus we created an **alias** here.

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

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

@Injectable()
export class CatsService {
  constructor(
    @Inject('CATS_REPOSITORY')
    private catsRepository: typeof Cat
  ) {}

  async findAll(): Promise<Cat[]> {
    return this.catsRepository.findAll<Cat>();
  }
}
```

The database connection is **asynchronous**, but Nest makes this process completely invisible for the end-user. The `CATS_REPOSITORY` provider is waiting for the db connection, and the `CatsService` is delayed until repository 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`.