SQL (TypeORM)

This chapter applies only to TypeScript

> **Warning** In this article, you'll learn how to create a `DatabaseModule` based on the **TypeORM** package from scratch using custom providers mechanism. 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/typeorm` package. To learn more, see here.

TypeORM is definitely the most mature Object Relational Mapper (ORM) available in the node.js world. Since it's written in TypeScript, it works pretty well with the Nest framework.

Getting started

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

```
$ npm install --save typeorm mysql2
```

The first step we need to do is to establish the connection with our database using `new DataSource().initialize()` class imported from the `typeorm` package. The `initialize()` function returns a `Promise`, and therefore we have to create an async provider.

```
@@filename(database.providers)
import { DataSource } from 'typeorm';

export const databaseProviders = [
  {
    provide: 'DATA_SOURCE',
    useFactory: async () => {
      const dataSource = new DataSource({
        type: 'mysql',
        host: 'localhost',
        port: 3306,
        username: 'root',
        password: 'root',
        database: 'test',
        entities: [
            __dirname + '/../**/*.entity{.ts,.js}',
        ],
        synchronize: true,
      });

      return dataSource.initialize();
    },
  },
];
```

> warning **Warning** Setting `synchronize: true` shouldn't be used in production - otherwise you can lose production data.

> 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 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 `DATA_SOURCE` object using `@Inject()` decorator. Each class that would depend on the `DATA_SOURCE` async provider will wait until a `Promise` is resolved.

Repository pattern

The TypeORM supports the repository design pattern, thus each entity has its own Repository. These repositories can be obtained from the database connection.

But firstly, we need at least one entity. We are going to reuse the `Photo` entity from the official documentation.

```
@@filename(photo.entity)
import { Entity, Column, PrimaryGeneratedColumn } from 'typeorm';

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

  @Column({ length: 500 })
  name: string;

  @Column('text')
  description: string;

  @Column()
  filename: string;

  @Column('int')
  views: number;

  @Column()
```

```
  isPublished: boolean;
}
```

The `Photo` entity belongs to the `photo` directory. This directory represents the `PhotoModule`. Now, let's create a **Repository** provider:

```
@@filename(photo.providers)
import { DataSource } from 'typeorm';
import { Photo } from './photo.entity';

export const photoProviders = [
  {
    provide: 'PHOTO_REPOSITORY',
    useFactory: (dataSource: DataSource) =>
dataSource.getRepository(Photo),
    inject: ['DATA_SOURCE'],
  },
];
```

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

Now we can inject the `Repository<Photo>` to the `PhotoService` using the `@Inject()` decorator:

```
@@filename(photo.service)
import { Injectable, Inject } from '@nestjs/common';
import { Repository } from 'typeorm';
import { Photo } from './photo.entity';

@Injectable()
export class PhotoService {
  constructor(
    @Inject('PHOTO_REPOSITORY')
    private photoRepository: Repository<Photo>,
  ) {}

  async findAll(): Promise<Photo[]> {
    return this.photoRepository.find();
  }
}
```

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

Here is a final `PhotoModule`:

```
@@filename(photo.module)
import { Module } from '@nestjs/common';
import { DatabaseModule } from '../database/database.module';
import { photoProviders } from './photo.providers';
import { PhotoService } from './photo.service';

@Module({
  imports: [DatabaseModule],
  providers: [
    ...photoProviders,
    PhotoService,
  ],
})
export class PhotoModule {}
```

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