Database (TypeORM)

In order to reduce a boilerplate necessary to start the adventure with any database, Nest comes with the ready to use `@nestjs/typeorm` package. We have selected **TypeORM** because it's definitely the most mature Object Relational Mapper (ORM) available so far. Since it's written in TypeScript, it works pretty well with the Nest framework.

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

```
$ npm install --save @nestjs/typeorm typeorm mysql
```

> **NOTICE**
> In this chapter we'll use a MySQL database, but **TypeORM** provides a support for a lot of different databases such as PostgreSQL, SQLite, and even MongoDB (NoSQL).

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

app.module.ts JS

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

@Module({
  imports: [
    TypeOrmModule.forRoot({
      type: 'mysql',
      host: 'localhost',
      port: 3306,
      username: 'root',
      password: 'root',
      database: 'test',
      entities: [__dirname + '/**/*.entity{.ts,.js}'],
      synchronize: true,
    }),
  ],
})
export class ApplicationModule {}
```

The `forRoot()` method accepts the same configuration object as `createConnection()` from the **TypeORM** package.

The `forRoot()` method accepts the same configuration object as `createConnection()` from the TypeORM package. Futhermore, instead of passing anything to `forRoot()`, we can create an `ormconfig.json` file in the project root directory.

```
{
  "type": "mysql",
  "host": "localhost",
  "port": 3306,
  "username": "root",
  "password": "root",
  "database": "test",
  "entities": ["src/**/*.entity{.ts,.js}"],
  "synchronize": true
}
```

Then, we can simply leave the parenthesis empty:

app.module.ts JS

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

@Module({
  imports: [TypeOrmModule.forRoot()],
})
export class ApplicationModule {}
```

Afterward, the `Connection` and `EntityManager` will be available to inject across entire project (without importing any module elsewhere), for example, in this way:

app.module.ts JS

```
import { Connection } from 'typeorm';

@Module({
  imports: [TypeOrmModule.forRoot(), PhotoModule],
})
export class ApplicationModule {
  constructor(private readonly connection: Connection) {}
}
```

Repository pattern

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

Firstly, we need at least one entity. We're gonna reuse the `Photo` entity from the official documentation.

photo.entity.ts JS

```
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`. It's your decision where you're gonna keep your model files. From our point of view, the best way's to hold them near their **domain**, in the corresponding module directory.

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

photo.module.ts JS

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { PhotoService } from './photo.service';
import { PhotoController } from './photo.controller';
```

```
import { Photo } from './photo.entity';

@Module({
  imports: [TypeOrmModule.forFeature([Photo])],
  providers: [PhotoService],
  controllers: [PhotoController],
})
export class PhotoModule {}
```

This module uses `forFeature()` method to define which repositories shall be registered in the current scope. Thanks to that we can inject the `PhotoRepository` to the `PhotoService` using the `@InjectRepository()` decorator:

photo.service.ts JS

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { Photo } from './photo.entity';

@Injectable()
export class PhotoService {
  constructor(
    @InjectRepository(Photo)
    private readonly photoRepository: Repository<Photo>,
  ) {}

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

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

Multiple databases

Some of your projects may require multiple database connections. Fortunately, this can also be achieved with this module. To work with multiple connections, the first thing to do is to create those connections. In this case, the connection naming becomes **mandatory**.

Say you have a `Person` entity and an `Album` entity, each stored in their own database.

```
const defaultOptions = {
  type: 'postgres',
  port: 5432,
  username: 'user',
  password: 'password',
  database: 'db',
  synchronize: true,
};

@Module({
  imports: [
    TypeOrmModule.forRoot({
      ...defaultOptions,
      host: 'photo_db_host',
      entities: [Photo],
    }),
    TypeOrmModule.forRoot({
      ...defaultOptions,
      name: 'personsConnection',
      host: 'person_db_host',
      entities: [Person],
    }),
    TypeOrmModule.forRoot({
      ...defaultOptions,
      name: 'albumsConnection',
      host: 'album_db_host',
      entities: [Album],
    }),
  ],
})
export class ApplicationModule {}
```

NOTICE

If you don't set any `name` for a connection, its name is set to `default`. Please note that you shouldn't have multiple connections without a name, or with the same name, otherwise they simply get overridden.

At this point, you have each of your `Photo`, `Person` and `Album` entities registered in their own connection. With this setup, you have to tell the `TypeOrmModule.forFeature()` function and the `@InjectRepository()` decorator which connection should be used. If you do not pass any connection name, the `default` connection is used.

```
@Module({
  imports: [
    TypeOrmModule.forFeature([Photo]),
```

```
    TypeOrmModule.forFeature([Person], 'personsConnection'),
    TypeOrmModule.forFeature([Album], 'albumsConnection'),
  ],
})
export class ApplicationModule {}
```

You can also inject the `Connection` or `EntityManager` for a given connection:

```
@Injectable()
export class PersonService {
  constructor(
    @InjectConnection('personsConnection')
    private readonly connection: Connection,
    @InjectEntityManager('personsConnection')
    private readonly entityManager: EntityManager,
  ) {}
}
```

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 repositories that are pulled from the connection instance. What then? The solution is to create fake repositories. In order to achieve that, we should set up **custom providers**. In fact, each registered repository is represented by a `EntityNameRepository` token, where `EntityName` is a name of your entity class.

The `@nestjs/typeorm` package exposes the `getRepositoryToken()` function which returns a prepared token based on a given entity.

```
@Module({
  providers: [
    PhotoService,
    {
      provide: getRepositoryToken(Photo),
      useValue: mockRepository,
    },
  ],
})
export class PhotoModule {}
```

Now a hardcoded `mockRepository` will be used as a `PhotoRepository`. Whenever any provider asks for `PhotoRepository` using an `@InjectRepository()` decorator, Nest will use a registered `mockRepository` object.

`PhotoRepository` using an `@InjectRepository()` decorator, Nest will use a registered `mockRepository` object.

Custom repository

TypeORM provides a feature called **custom repositories**. To learn more about it, visit **this** page. Basically, custom repositories allow you to extend a base repository class, and enrich it with a couple of special methods.

In order to create your custom repository, use the `@EntityRepository()` decorator and extend the `Repository` class.

```
@EntityRepository(Author)
export class AuthorRepository extends Repository<Author> {}
```

> **HINT**
> Both `@EntityRepository()` and `Repository` are exposed from `typeorm` package.

Once the class is created, the next step is to hand over the instantiation responsibility to Nest. For this, we have to pass `AuthorRepository` class to the `TypeOrm.forFeature()` method.

```
@Module({
  imports: [TypeOrmModule.forFeature([AuthorRepository])],
  controller: [AuthorController],
  providers: [AuthorService],
})
export class AuthorModule {}
```

Afterward, simply inject the repository using the following construction:

```
@Injectable()
export class AuthorService {
  constructor(private readonly authorRepository: AuthorRepository) {}
}
```

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:

```
TypeOrmModule.forRootAsync({
  useFactory: () => ({
    type: 'mysql',
    host: 'localhost',
    port: 3306,
    username: 'root',
    password: 'root',
    database: 'test',
    entities: [__dirname + '/**/*.entity{.ts,.js}'],
    synchronize: true,
  }),
});
```

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

```
TypeOrmModule.forRootAsync({
  imports: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
    type: 'mysql',
    host: configService.getString('HOST'),
    port: configService.getString('PORT'),
    username: configService.getString('USERNAME'),
    password: configService.getString('PASSWORD'),
    database: configService.getString('DATABASE'),
    entities: [__dirname + '/**/*.entity{.ts,.js}'],
    synchronize: true,
  }),
  inject: [ConfigService],
});
```

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

```
TypeOrmModule.forRootAsync({
  useClass: TypeOrmConfigService,
});
```

Above construction will instantiate `TypeOrmConfigService` inside `TypeOrmModule` and will leverage it to create options object. The `TypeOrmConfigService` has to implement `TypeOrmOptionsFactory` interface.

```
@Injectable()
class TypeOrmConfigService implements TypeOrmOptionsFactory {
  createTypeOrmOptions(): TypeOrmModuleOptions {
    return {
      type: 'mysql',
      host: 'localhost',
      port: 3306,
      username: 'root',
      password: 'root',
      database: 'test',
      entities: [__dirname + '/**/*.entity{.ts,.js}'],
      synchronize: true,
    };
  }
}
```

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

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

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

Become a sponsor