Nest

Integración con TypeORM en MySQL

CertiDevs

Índice de contenidos

1. Integración con TypeORM en MySQL

TypeORM es un **ORM** (Object Relational Mapper) para TypeScript y JavaScript que facilita la interacción con bases de datos relacionales como MySQL.

Integración con TypeORM en MySQL y NestJS

NestJS tiene una integración nativa con TypeORM, lo que simplifica el trabajo con bases de datos en tus aplicaciones.

2. Instalación y configuración

Para comenzar, instala TypeORM, el paquete de NestJS para TypeORM y el paquete del controlador de MySQL:

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

Luego, importa el **módulo** `TypeOrmModule` en tu módulo principal (`AppModule`), y configura la conexión a la base de datos utilizando la **función** `forRoot()`:

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

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

En este ejemplo, se configura una conexión a MySQL con las credenciales y opciones apropiadas.

La propiedad `entities` especifica la ubicación de tus clases de entidad, y `synchronize`: `true` habilita la sincronización automática del esquema de la base de datos con tus entidades.

3. Creación de entidades

Las **entidades** en TypeORM son **clases** que representan **tablas** en la base de datos y sus relaciones.

Para **crear una entidad**, define una **clase** TypeScript decorada con el decorador `@Entity()` y proporciona propiedades con decoradores de columna, como `@PrimaryGeneratedColumn()`, `@Column()` y `@CreateDateColumn()`.

Por ejemplo, aquí hay una entidad básica llamada `Cat`:

```
import {
Entity,
PrimaryGeneratedColumn,
Column,
CreateDateColumn,
} from 'typeorm';

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

    @Column()
    name: string;

    @Column()
    age: number;

    @Column()
    breed: string;

    @CreateDateColumn()
    createdAt: Date;
}
```

En este ejemplo, `Cat` representa una tabla en la base de datos con columnas para `id`, `name`, `age`, `breed` y `createdAt`.

4. Repositorios y servicios

TypeORM proporciona **repositorios** para acceder y manipular datos en la base de datos.

Para utilizar un **repositorio**, importa el módulo `TypeOrmModule` en el módulo donde deseas utilizarlo e indica la entidad correspondiente utilizando la función `forFeature()`:

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { CatsController } from './cats.controller';
```

```
import { CatsService } from './cats.service';
import { Cat } from './cat.entity';

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

Luego, inyecta el repositorio en tu servicio utilizando el decorador `@InjectRepository()` y la clase de entidad:

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

@Injectable()
export class CatsService {

    constructor(@InjectRepository(Cat) private readonly catRepository: Repository<Cat
>) {}

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

    async create(cat: Cat): Promise<Cat> {
        return await this.catRepository.save(cat);
    }

    async findOne(id: number): Promise<Cat> {
        return await this.catRepository.findOne(id);
    }

    async update(id: number, cat: Cat): Promise<void> {
        await this.catRepository.update(id, cat);
    }

    async delete(id: number): Promise<void> {
        await this.catRepository.delete(id);
    }
}
```

En este ejemplo, `CatsService` utiliza el repositorio `catRepository` para realizar operaciones CRUD en la tabla `Cat` de la base de datos.

Gracias a la **inyección de dependencias**, el repositorio se maneja automáticamente por NestJS y

TypeORM.

5. Uso en controladores

Ahora que tienes un servicio que utiliza TypeORM para interactuar con la base de datos, puedes utilizar este servicio en tus **controladores**.

Por ejemplo, en `CatsController`:

```
import { Controller, Get, Post, Body, Param, Put, Delete } from '@nestjs/common';
import { CatsService } from './cats.service';
import { Cat } from './cat.entity';

@Controller('cats')
export class CatsController {
    constructor(private readonly catsService: CatsService) {}

    @Get()
    async findAll(): Promise<Cat[]> {
        return await this.catsService.findAll();
    }

    @Post()
    async create(@Body() cat: Cat): Promise<Cat> {
        return await this.catsService.create(cat);
    }

    @Get(':id')
    async findOne(@Param('id') id: number): Promise<Cat> {
        return await this.catsService.findOne(id);
    }

    @Put(':id')
    async update(@Param('id') id: number, @Body() cat: Cat): Promise<void> {
        await this.catsService.update(id, cat);
    }

    @Delete(':id')
    async delete(@Param('id') id: number): Promise<void> {
        await this.catsService.delete(id);
    }
}
```

En este ejemplo, `CatsController` utiliza `CatsService` para manejar las solicitudes HTTP y realizar **operaciones CRUD** en la base de datos.

6. Asociaciones entre entidades

Para crear **entidades con asociaciones** en TypeORM, puedes utilizar los decoradores que representan las relaciones entre entidades.

Veamos un ejemplo de cada tipo de asociación.

6.1. One-to-One

Supongamos que tenemos una **entidad** `User` y otra **entidad** `Profile`.

Cada usuario tiene un perfil único y cada perfil pertenece a un único usuario.

Para representar esta relación, utilizamos los decoradores `@OneToOne` y `@JoinColumn`.

Archivo `user.entity.ts`:

```
import { Entity, PrimaryGeneratedColumn, Column, OneToOne, JoinColumn } from 'typeorm
';
import { Profile } from './profile.entity';

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

    @Column()
    username: string;

    @OneToOne(() => Profile, (profile) => profile.user, { cascade: true })
    @JoinColumn()
    profile: Profile;
}
```

Archivo `profile.entity.ts`:

```
import { Entity, PrimaryGeneratedColumn, Column, OneToOne } from 'typeorm';
import { User } from './user.entity';

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

    @Column()
    bio: string;

    @OneToOne(() => User, (user) => user.profile)
```

```
    user: User;
}
```

En este ejemplo, la entidad `User` tiene una relación `one-to-one` con la entidad `Profile`.

El **decorador** `@JoinColumn` indica que la entidad `User` es la propietaria de la relación y que la tabla `user` contendrá la **clave foránea** de `profile`.

6.2. One-to-Many y Many-to-One

Imagina que ahora tienes una entidad `Post` y quieres representar que un usuario puede tener múltiples publicaciones, pero cada publicación solo pertenece a un usuario.

Para lograr esto, utilizamos los decoradores `@OneToMany` y `@ManyToOne`.

Archivo `user.entity.ts`:

```
import { Entity, PrimaryGeneratedColumn, Column, OneToMany } from 'typeorm';
import { Post } from './post.entity';

@Entity()
export class User {
  // ...

  @OneToMany(() => Post, (post) => post.user, { cascade: true })
  posts: Post[];
}
```

Archivo `post.entity.ts`:

```
import { Entity, PrimaryGeneratedColumn, Column, ManyToOne } from 'typeorm';
import { User } from './user.entity';

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

    @Column()
    title: string;

    @Column()
    content: string;

    @ManyToOne(() => User, (user) => user.posts)
    user: User;
}
```

En este ejemplo, la entidad `User` tiene una relación `one-to-many` con la entidad `Post`.

La propiedad `posts` en `User` contiene un array de objetos `Post`.

Por otro lado, la entidad Post tiene una relación `many-to-one` con la entidad `User`.

6.3. Many-to-Many

Supongamos que ahora también tenemos una entidad `Tag` y queremos representar que una publicación puede tener múltiples etiquetas y una etiqueta puede estar asociada a múltiples publicaciones.

Para representar esta relación, utilizamos los decoradores `@ManyToMany` y `@JoinTable`.

Archivo `post.entity.ts`:

```
import { Entity, PrimaryGeneratedColumn, Column, ManyToMany, JoinTable } from '
typeorm';
import { Tag } from './tag.entity';

@Entity()
export class Post {
    // ...

    @ManyToMany(() => Tag, (tag) => tag.posts, { cascade: true })
    @JoinTable()
    tags: Tag[];
}
```

Archivo `tag.entity.ts`:

```
import { Entity, PrimaryGeneratedColumn, Column, ManyToMany } from 'typeorm';
import { Post } from './post.entity';

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

    @Column()
    name: string;

    @ManyToMany(() => Post, (post) => post.tags)
    posts: Post[];
}
```

En este ejemplo, la entidad `Post` tiene una relación `many-to-many` con la entidad `Tag`.

La propiedad `tags` en `Post` contiene un array de objetos `Tag`.

Por otro lado, la entidad `Tag` también tiene una relación `many-to-many` con la entidad `Post`.

El decorador `@JoinTable()` indica que la entidad `Post` es la **propietaria** de la relación y que se generará una **tabla de unión (intermedia)** para manejar la relación many-to-many.

Al utilizar estas relaciones en tus entidades y configurar TypeORM con `synchronize: true`, las tablas de la base de datos se generarán automáticamente a partir de las entidades y sus relaciones.

Ten en cuenta que, en un entorno de producción, es recomendable desactivar la sincronización automática y gestionar el esquema de la base de datos mediante migraciones. Esto te permitirá tener un mayor control sobre los cambios en la estructura de la base de datos y evitar posibles pérdidas de datos no deseadas.