School Management

please check the [api design](./API Design.md)

ENV Configuration

A good approach for using this technique in Nest is to create a `ConfigModule` that exposes a `ConfigService` which loads the appropriate `.env` file. While you may choose to write such a module yourself, for convenience Nest provides the `@nestjs/config` package out-of-the box. We'll cover this package in the current chapter.

```
npm i --save @nestjs/config
```

Config app.module.ts

Typically, we'll import it into the root `AppModule` and control its behavior using the `.forRoot()` static method. During this step, environment variable key/value pairs are parsed and resolved.

`app.module.ts`

```
import { Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';

@Module({
  imports: [
    ConfigModule.forRoot({
            isGlobal: true,
            envFilePath: '.env',
     }),
    ],
})
export class AppModule {}
```

using `main.ts` with .env

```
import { ConfigService } from '@nestjs/config';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe());

  const configService = app.get(ConfigService);
  const PORT = configService.getOrThrow<number>('PORT');
  await app.listen(PORT);
}
bootstrap();
```

install packages

```
pnpm add @nestjs/typeorm typeorm pg @types/pg
```

TypeORM Integration

For integrating with SQL and NoSQL databases, Nest provides the `@nestjs/typeorm` package. TypeORM is the most mature Object Relational Mapper (ORM) available for TypeScript. TypeORM provides support for many relational databases, such as `PostgreSQL`, `Oracle`, `Microsoft SQL Server`, `SQLite`, and even `NoSQL`

create a `database.module.ts`

```
nest g mo database
```

add below code to `database.module.ts`

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { ConfigModule, ConfigService } from '@nestjs/config';

@Module({
  imports: [
    ConfigModule,
    TypeOrmModule.forRootAsync({
      imports: [ConfigModule],
      useFactory: (configService: ConfigService) => ({
        type: 'postgres',
        host: configService.getOrThrow<string>('DB_HOST'),
        port: configService.getOrThrow<number>('DB_PORT'),
        username: configService.getOrThrow<string>('DB_USERNAME'),
        password: configService.getOrThrow<string>('DB_PASSWORD'),
        database: configService.getOrThrow<string>('DB_NAME'),
        entities: [__dirname + '/../**/*.entity{.ts,.js}'],
        synchronize: configService.getOrThrow<boolean>('DB_SYNC', true),
        logging: configService.getOrThrow<boolean>('DB_LOGGING', false),
        migrations: [__dirname + '/../migrations/**/*{.ts,.js}'],
      }),
      inject: [ConfigService],
    }),
  ],
})
export class DatabaseModule {}
```

Repository pattern

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

one-to-one relation

Here, we are using a new decorator called `@OneToOne`. It allows us to create a one-to-one relationship between two entities.

We also add a `@JoinColumn` decorator, which indicates that this side of the relationship will own the relationship. Relations can be unidirectional or bidirectional. Only one side of relational can be owning. Using `@JoinColumn` decorator is required on the owner side of the relationship.

Points to note

1. The Student entity owns the relationship (has the foreign key column via `@JoinColumn())`
2. You can add `cascade` and `delete` behavior for better data integrity

`student.entity.ts`

```
import {
  Entity,
  PrimaryGeneratedColumn,
  Column,
  OneToOne,
  // ManyToMany,
  // JoinTable,
  JoinColumn,
  Relation,
} from 'typeorm';
import { Profile } from '../../profiles/entities/profile.entity';
// import { Course } from './course.entity';

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

  @Column('date')
  enrollmentDate: string;

  @Column({ nullable: true })
  degreeProgram: string;

  @Column({ type: 'decimal', precision: 3, scale: 2, nullable: true })
  gpa: number;

  @Column({ type: 'timestamp', default: () => 'CURRENT_TIMESTAMP' })
  createdAt: Date;
  @Column({
    type: 'timestamp',
    default: () => 'CURRENT_TIMESTAMP',
    onUpdate: 'CURRENT_TIMESTAMP',
  })
  updatedAt: Date;

  @OneToOne(() => Profile, (profile) => profile.id, {
    cascade: true,
    onDelete: 'CASCADE',
  })
  @JoinColumn()
  profile: Relation<Profile>;

  //   @ManyToMany(() => Course)
  //   @JoinTable() // Define the join table for the many-to-many relationship
  //   courses: Course[];
}
```

> If you use ESM in your TypeScript project, you should use the `Relation` wrapper type in relation properties to avoid circular dependency issues. Let's modify our entities:

To use this entity we must register it in the `student.module.ts`. Also we need to import the `DatabaseModule` since we will need it to use in the `student.service.ts`

`student.module.ts`

```
import { Module } from '@nestjs/common';
import { StudentsService } from './students.service';
import { StudentsController } from './students.controller';
import { DatabaseModule } from 'src/database/database.module';
import { Student } from './entities/student.entity';
import { TypeOrmModule } from '@nestjs/typeorm';
import { Profile } from 'src/profiles/entities/profile.entity';

@Module({
  imports: [DatabaseModule, TypeOrmModule.forFeature([Student, Profile])],
  controllers: [StudentsController],
  providers: [StudentsService],
})
export class StudentsModule {}
```

Inverse side of the relationship (Profile)

Relations can be `unidirectional` or `bidirectional`. Currently, our relation between `student` and `profile`. The owner of the relation is student and profile doesn't know anything about student. We need to add an `inverse relation`, and make relations between Profile and Student bidirectional profile.entity.ts

```
import {
  Entity,
  PrimaryGeneratedColumn,
  Column,
  OneToOne,
  Relation,
} from 'typeorm';
import { Student } from '../../students/entities/student.entity';

export enum Role {
  STUDENT = 'student',
  FACULTY = 'faculty',
  ADMIN = 'admin',
  GUEST = 'guest',
}

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

  @Column()
  firstName: string;

  @Column()
  lastName: string;

  @Column()
  email: string;

  @Column({ type: 'enum', enum: Role, default: Role.GUEST })
  role: Role;

  @Column({ type: 'timestamp', default: () => 'CURRENT_TIMESTAMP' })
  createdAt: Date;

  @Column({
    type: 'timestamp',
    default: () => 'CURRENT_TIMESTAMP',
    onUpdate: 'CURRENT_TIMESTAMP',
  })
  updatedAt: Date;

  @OneToOne(() => Student, (student) => student.profile)
  student: Relation<Student>;
}
```

To use this entity we must register it in the `profile.module.ts`. Also we need to import the `DatabaseModule` since we will need it to use in the `profile.service.ts`

`profiles.module.ts`

```
import { Module } from '@nestjs/common';
import { ProfilesService } from './profiles.service';
import { ProfilesController } from './profiles.controller';
import { DatabaseModule } from 'src/database/database.module';
import { TypeOrmModule } from '@nestjs/typeorm';
import { Profile } from './entities/profile.entity';

@Module({
  imports: [DatabaseModule, TypeOrmModule.forFeature([Profile])],
  controllers: [ProfilesController],
  providers: [ProfilesService],
})
export class ProfileModule {}
```

`profiles.controller.ts`

```
import {
  Controller,
  Get,
  Post,
  Body,
  Patch,
  Param,
  Delete,
  ParseIntPipe,
  Query,
} from '@nestjs/common';
import { ProfilesService } from './profiles.service';
import { CreateProfileDto, UpdateProfileDto } from './dto';

@Controller('profiles')
export class ProfilesController {
  constructor(private readonly profilesService: ProfilesService) {}

  @Post()
  create(@Body() createProfileDto: CreateProfileDto) {
    return this.profilesService.create(createProfileDto);
  }

  @Get()
  findAll(@Query('email') email?: string) {
    return this.profilesService.findAll(email);
  }
  @Get(':id')
  findOne(@Param('id', ParseIntPipe) id: number) {
    return this.profilesService.findOne(id);
  }

  @Patch(':id')
  update(
    @Param('id', ParseIntPipe) id: number,
    @Body() updateProfileDto: UpdateProfileDto,
  ) {
    return this.profilesService.update(id, updateProfileDto);
  }

  @Delete(':id')
  remove(@Param('id', ParseIntPipe) id: number) {
    return this.profilesService.remove(id);
  }
}
```

`profiles.service.ts`

```
import { Injectable } from '@nestjs/common';
import { CreateProfileDto } from './dto/create-profile.dto';
import { UpdateProfileDto } from './dto/update-profile.dto';
import { InjectRepository } from '@nestjs/typeorm';
import { Profile } from './entities/profile.entity';
import { Repository } from 'typeorm';

@Injectable()
export class ProfilesService {
  constructor(
    @InjectRepository(Profile) private profileRepository: Repository<Profile>,
  ) {}

  async create(createProfileDto: CreateProfileDto): Promise<Profile> {
    return await this.profileRepository
      .save(createProfileDto)
      .then((profile) => {
        return profile;
      })
      .catch((error) => {
        console.error('Error creating profile:', error);
        throw new Error('Failed to create profile');
      });
  }

  async findAll(email?: string) {
    if (email) {
      return await this.profileRepository.find({
        where: {
          email: email,
        },
        relations: ['student'], // Ensure to load the student relation
      });
    }
    return this.profileRepository.find({
      relations: ['student'], // Ensure to load the student relation
    });
  }

  async findOne(id: number): Promise<Profile | string> {
    return await this.profileRepository
      .findOneBy({ id })
      .then((profile) => {
        if (!profile) {
          return `No profile found with id ${id}`;
        }
        return profile;
      })
      .catch((error) => {
        console.error('Error finding profile:', error);
        throw new Error(`Failed to find profile with id ${id}`);
      });
  }

  async update(
    id: number,
    updateProfileDto: UpdateProfileDto,
  ): Promise<Profile | string> {
    await this.profileRepository.update(id, updateProfileDto);

    return await this.findOne(id);
  }

  async remove(id: number): Promise<string> {
    return await this.profileRepository
      .delete(id)
      .then((result) => {
        if (result.affected === 0) {
          return `No profile found with id ${id}`;
        }
        return `Profile with id ${id} has been removed`;
      })
      .catch((error) => {
        console.error('Error removing profile:', error);
        throw new Error(`Failed to remove profile with id ${id}`);
      });
  }
}
```

`students.service.ts`

```
import { Injectable, NotFoundException } from '@nestjs/common';
import { CreateStudentDto, UpdateStudentDto } from './dto';
import { InjectRepository } from '@nestjs/typeorm';
import { Student } from './entities/student.entity';
import { Repository } from 'typeorm';
import { Profile } from 'src/profiles/entities/profile.entity';

@Injectable()
export class StudentsService {
  constructor(
    @InjectRepository(Student) private studentRepository: Repository<Student>,
    @InjectRepository(Profile) private profileRepository: Repository<Profile>,
  ) {}

  async create(createStudentDto: CreateStudentDto): Promise<Student> {
    // if profile id exists, we need to check if the profile is already associated
with a student
    const existingProfile = await this.profileRepository.findOneBy({
      id: createStudentDto.profileId,
    });

    if (!existingProfile) {
      throw new NotFoundException(
        `Profile with ID ${createStudentDto.profileId} not found`,
      );
    }

    return this.studentRepository.save(createStudentDto);
  }

  async findAll(name?: string): Promise<Student[] | Student> {
    if (name) {
      return await this.studentRepository.find({
        where: {
          profile: {
            firstName: name,
          },
        },
        relations: ['profile'], // Ensure to load the profile relation
      });
    }
    return await this.studentRepository.find({
      relations: ['profile'], // Ensure to load the profile relation
    });
  }

  async findOne(id: number): Promise<Student | string> {
    return await this.studentRepository
      .findOneBy({ id })
      .then((student) => {
        if (!student) {
          return `No student found with id ${id}`;
        }
```

```
          return student;
        })
        .catch((error) => {
          console.error('Error finding student:', error);
          throw new Error(`Failed to find student with id ${id}`);
        });
    }

    async update(id: number, updateStudentDto: UpdateStudentDto) {
      return await this.studentRepository
        .update(id, updateStudentDto)
        .then((result) => {
          if (result.affected === 0) {
            return `No student found with id ${id}`;
          }
        })
        .catch((error) => {
          console.error('Error updating student:', error);
          throw new Error(`Failed to update student with id ${id}`);
        });
    }

    async remove(id: number): Promise<string> {
      return await this.studentRepository
        .delete(id)
        .then((result) => {
          if (result.affected === 0) {
            return `No student found with id ${id}`;
          }
          return `Student with id ${id} has been removed`;
        })
        .catch((error) => {
          console.error('Error removing student:', error);
          throw new Error(`Failed to remove student with id ${id}`);
        });
    }
  }
```

students.controller.ts

```
import { Injectable } from '@nestjs/common';
import { CreateProfileDto } from './dto/create-profile.dto';
import { UpdateProfileDto } from './dto/update-profile.dto';
import { InjectRepository } from '@nestjs/typeorm';
import { Profile } from './entities/profile.entity';
import { Repository } from 'typeorm';

@Injectable()
export class ProfilesService {
  constructor(
    @InjectRepository(Profile) private profileRepository: Repository<Profile>,
  ) {}

  async create(createProfileDto: CreateProfileDto) {
    return await this.profileRepository
      .save(createProfileDto)
      .then((profile) => {
        return `Profile with id ${profile.id} has been created`;
      })
      .catch((error) => {
        console.error('Error creating profile:', error);
        throw new Error('Failed to create profile');
      });
  }

  async findAll(email?: string) {
    if (email) {
      return await this.profileRepository.find({
        where: {
          email: email,
        },
        relations: ['student'], // Ensure to load the student relation
      });
    }
    return this.profileRepository.find({
      relations: ['student'], // Ensure to load the student relation
    });
  }

  async findOne(id: number) {
    return await this.profileRepository
      .findOneBy({ id })
      .then((profile) => {
        if (!profile) {
          return `No profile found with id ${id}`;
        }
        return profile;
      })
      .catch((error) => {
        console.error('Error finding profile:', error);
        throw new Error(`Failed to find profile with id ${id}`);
      });
  }

  async update(
    id: number,
    updateProfileDto: UpdateProfileDto,
  ): Promise<string> {
    return await this.profileRepository
      .update(id, updateProfileDto)
      .then(() => {
        return `Profile with id ${id} has been updated`;
      })
      .catch((error) => {
        console.error('Error updating profile:', error);
```

```
          throw new Error(`Failed to update profile with id ${id}`);
        });
    }

    async remove(id: number): Promise<string> {
      return await this.profileRepository
        .delete(id)
        .then((result) => {
          if (result.affected === 0) {
            return `No profile found with id ${id}`;
          }
          return `Profile with id ${id} has been removed`;
        })
        .catch((error) => {
          console.error('Error removing profile:', error);
          throw new Error(`Failed to remove profile with id ${id}`);
        });
    }
  }
```

Creating a many-to-one / one-to-many relation

Let's create a many-to-one/one-to-many relation. Let's say a **Department** ↔ **Course** : Many-to-One relationship. A department can have many courses, but each course belongs to a single department.

lets create a `course.entity.ts`

> Course entity has `manyToOne()` relationship meaning its the owner of this relationship hence it will store the id of the related object

```
import {
  Entity,
  PrimaryGeneratedColumn,
  Column,
  ManyToOne,
  OneToMany,
} from 'typeorm';
import { Department } from '../../departments/entities/department.entity';
// import { Lecture } from './lecture.entity';

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

  @Column()
  title: string;

  @Column({ nullable: true })
  description: string;
  @Column('int')
  credits: number;

  @Column({ nullable: true })
  duration: string;

  @Column('date', { nullable: true })
  startDate: string;

  @Column('date', { nullable: true })
  endDate: string;

  @Column({ type: 'timestamp', default: () => 'CURRENT_TIMESTAMP' })
  createdAt: Date;

  @Column({
    type: 'timestamp',
    default: () => 'CURRENT_TIMESTAMP',
    onUpdate: 'CURRENT_TIMESTAMP',
  })
  updatedAt: Date;

  @ManyToOne(() => Department, (department) => department.id)
  department: Department['id'];

  // @OneToMany(() => Lecture, (lecture) => lecture.course)
  // lectures: Lecture[]; // store only the department ID
}
```

let's create the other side of the relationship that is:

`department.entity.ts`

```
import { Entity, PrimaryGeneratedColumn, Column, OneToMany } from 'typeorm';
import { Course } from '../../courses/entities/course.entity';

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

  @Column()
  name: string;

  @Column({ nullable: true })
  description: string;

  @Column({ nullable: true })
  headOfDepartment: string;
```

```
    @Column({ type: 'timestamp', default: () => 'CURRENT_TIMESTAMP' })
    createdAt: Date;

    @Column({
      type: 'timestamp',
      default: () => 'CURRENT_TIMESTAMP',
      onUpdate: 'CURRENT_TIMESTAMP',
    })
    updatedAt: Date;

      @OneToMany(() => Course, (course) => course.department) // Define the
  relationship with Course by specifying the inverse side
    courses: Course[]; // Store the related courses in an array
    // This will allow you to access all courses related to this department
  }
```

Remember to register/ import `DatabaseModule` and `TypeOrmModule.forFeature([Department])` in the `DepartmentsModule`

`department.module.ts`

```
import { Module } from '@nestjs/common';
import { DepartmentsService } from './departments.service';
import { DepartmentsController } from './departments.controller';
import { DatabaseModule } from 'src/database/database.module';
import { TypeOrmModule } from '@nestjs/typeorm';
import { Department } from './entities/department.entity';

@Module({
  imports: [DatabaseModule, TypeOrmModule.forFeature([Department])],
  controllers: [DepartmentsController],
  providers: [DepartmentsService],
})
export class DepartmentsModule {}
```

create the `departments.controller.ts`

```
import {
  Controller,
  Get,
  Post,
  Body,
  Patch,
  Param,
  Delete,
  ParseIntPipe,
  Query,
} from '@nestjs/common';
import { DepartmentsService } from './departments.service';
import { CreateDepartmentDto, UpdateDepartmentDto } from './dto';

@Controller('departments')
export class DepartmentsController {
  constructor(private readonly departmentsService: DepartmentsService) {}

  @Post()
  create(@Body() createDepartmentDto: CreateDepartmentDto) {
    return this.departmentsService.create(createDepartmentDto);
  }

  @Get()
  findAll(@Query('search') search?: string) {
    return this.departmentsService.findAll(search);
  }

  @Get(':id')
  findOne(@Param('id', ParseIntPipe) id: number) {
    return this.departmentsService.findOne(id);
  }

  @Patch(':id')
  update(
    @Param('id', ParseIntPipe) id: number,
    @Body() updateDepartmentDto: UpdateDepartmentDto,
  ) {
    return this.departmentsService.update(id, updateDepartmentDto);
  }

  @Delete(':id')
  remove(@Param('id', ParseIntPipe) id: number) {
    return this.departmentsService.remove(id);
  }
}
```

and `departments.service.ts`

```
import { Injectable } from '@nestjs/common';
import { CreateDepartmentDto } from './dto/create-department.dto';
import { UpdateDepartmentDto } from './dto/update-department.dto';
import { InjectRepository } from '@nestjs/typeorm';
import { Department } from './entities/department.entity';
import { Repository } from 'typeorm';

@Injectable()
export class DepartmentsService {
  constructor(
    @InjectRepository(Department)
    private departmentRepository: Repository<Department>,
  ) {}

  create(createDepartmentDto: CreateDepartmentDto) {
    return this.departmentRepository.save(createDepartmentDto);
  }

  findAll(search?: string) {
    if (search) {
      return this.departmentRepository.find({
        where: [{ name: `%${search}%` }, { description: `%${search}%` }],
        relations: ['courses'],
      });
    }
    return this.departmentRepository.find({
      relations: ['courses'],
    });
  }

  findOne(id: number) {
    return this.departmentRepository.findOne({
      where: { id },
      relations: ['courses'],
    });
  }

  update(id: number, updateDepartmentDto: UpdateDepartmentDto) {
    return this.departmentRepository.update(id, updateDepartmentDto);
  }

  remove(id: number) {
    return this.departmentRepository.delete(id);
  }
}
```

the same goes to `courses.module.ts` but this time add also register `Department` to the repository.

```
import { Module } from '@nestjs/common';
import { CoursesService } from './courses.service';
import { CoursesController } from './courses.controller';
import { Course } from './entities/course.entity';
import { TypeOrmModule } from '@nestjs/typeorm';
import { DatabaseModule } from 'src/database/database.module';
import { Department } from '../departments/entities/department.entity';

@Module({
  imports: [DatabaseModule, TypeOrmModule.forFeature([Course, Department])],
  providers: [CoursesService],
  controllers: [CoursesController],
})
export class CoursesModule {}
```

```
```

`courses.controller.ts`

```
import {
  Controller,
  Get,
  Post,
  Body,
  Patch,
  Param,
  Delete,
  ParseIntPipe,
  Query,
} from '@nestjs/common';
import { CoursesService } from './courses.service';
import { CreateCourseDto } from './dto/create-course.dto';
import { UpdateCourseDto } from './dto/update-course.dto';

@Controller('courses')
export class CoursesController {
  constructor(private readonly coursesService: CoursesService) {}

  // http://localhost:3000/courses
  @Post()
  create(@Body() createCourseDto: CreateCourseDto) {
    return this.coursesService.create(createCourseDto);
  }

  // http://localhost:3000/courses?search=Math
  @Get()
  findAll(@Query('search') search?: string) {
    return this.coursesService.findAll(search);
  }

  // http://localhost:3000/courses/1
  @Get(':id')
  findOne(@Param('id', ParseIntPipe) id: number) {
    return this.coursesService.findOne(id);
  }

  // http://localhost:3000/courses/1
  @Patch(':id')
  update(
    @Param('id', ParseIntPipe) id: number,
    @Body() updateCourseDto: UpdateCourseDto,
  ) {
    return this.coursesService.update(id, updateCourseDto);
  }

  // http://localhost:3000/courses/1
  @Delete(':id')
  remove(@Param('id', ParseIntPipe) id: number) {
    return this.coursesService.remove(id);
  }
}
```

`courses.service.ts`

```
import { Injectable, NotFoundException } from '@nestjs/common';
import { CreateCourseDto } from './dto/create-course.dto';
import { UpdateCourseDto } from './dto/update-course.dto';
import { InjectRepository } from '@nestjs/typeorm';
import { Course } from './entities/course.entity';
import { Repository, Like } from 'typeorm';
import { Department } from '../departments/entities/department.entity';

@Injectable()
export class CoursesService {
  constructor(
    @InjectRepository(Course) private courseRepository: Repository<Course>,
    @InjectRepository(Department)
    private departmentRepository: Repository<Department>,
  ) {}

  async create(createCourseDto: CreateCourseDto): Promise<Course> {
    // Find the department
    const department = await this.departmentRepository.findOne({
      where: { id: createCourseDto.departmentId },
    });

    if (!department) {
      throw new NotFoundException(
        `Department with ID ${createCourseDto.departmentId} not found`,
      );
    }

    // Create a new course instance
    const newCourse = this.courseRepository.create({
      title: createCourseDto.title,
      description: createCourseDto.description,
      credits: createCourseDto.credits,
      duration: createCourseDto.duration,
      startDate: createCourseDto.startDate,
      endDate: createCourseDto.endDate,
      department: createCourseDto.departmentId,
    });

    // Save the course to the database
    return this.courseRepository.save(newCourse);
  }

  async findAll(search?: string): Promise<Course[]> {
    if (search) {
      return this.courseRepository.find({
        where: [
          { title: Like(`%${search}%`) },
          { description: Like(`%${search}%`) },
        ],
        relations: ['department'],
      });
    }
    return this.courseRepository.find({
      relations: ['department'],
    });
  }

  async findOne(id: number): Promise<Course> {
    const course = await this.courseRepository.findOne({
      where: { id },
      relations: ['department'],
    });

    if (!course) {
      throw new NotFoundException(`Course with ID ${id} not found`);
    }

    return course;
  }

  async update(id: number, updateCourseDto: UpdateCourseDto): Promise<Course> {
    // First check if the course exists
    const course = await this.courseRepository.findOne({
      where: { id },
    });

    if (!course) {
      throw new NotFoundException(`Course with ID ${id} not found`);
    }

    // If departmentId is provided, find the department
    if (updateCourseDto.departmentId) {
      const departmentId = await this.departmentRepository.findOne({
        where: { id: updateCourseDto.departmentId },
      });

      if (!departmentId) {
        throw new NotFoundException(
          `Department with ID ${updateCourseDto.departmentId} not found`,
        );
      }
    }

    // Update the course
    await this.courseRepository.update(id, {
      title: updateCourseDto.title,
      description: updateCourseDto.description,
      credits: updateCourseDto.credits,
      duration: updateCourseDto.duration,
      startDate: updateCourseDto.startDate,
      endDate: updateCourseDto.endDate,
      department: updateCourseDto.departmentId,
    });

    // Return the updated course
    return this.findOne(id);
  }

  async remove(id: number): Promise<void> {
    const result = await this.courseRepository.delete(id);

    if (result.affected === 0) {
      throw new NotFoundException(`Course with ID ${id} not found`);
    }
  }
}
```

Creating a many-to-many relation

Student ↔ **Course** : A student can be enrolled in multiple courses, and each course can have many students.

In a many-to-many relationship, both entities can have multiple instances of each other. For example, a student can be enrolled in multiple courses, and each course can have multiple students enrolled in it. TypeORM makes it easy to establish this relationship with the `@ManyToMany` decorator.

Implementing the Many-to-Many Relationship

1. Update Student Entity

In the Student entity, we define the many-to-many relationship with Course entities:

```
import {
  Entity,
  // ...existing code...
  ManyToMany,
  JoinTable,
  // ...existing code...
} from 'typeorm';
import { Profile } from '../../profiles/entities/profile.entity';
import { Course } from 'src/courses/entities/course.entity';

@Entity()
export class Student {
  // ...existing code...

  @ManyToMany(() => Course, (course) => course.students)
  @JoinTable() // Important! This creates the join table in the database
  courses: Relation<Course[]>;
}
```

Key points:

- The `@ManyToMany()` decorator defines the relationship with Course entities
- The first parameter is a function returning the Course entity class
- The second parameter defines the inverse side of the relationship (how to reach Student from Course)
- The `@JoinTable()` decorator is required on ONE side of the relationship to specify which side owns the relationship and will create the join table

2. Update Course Entity

Similarly, we update the Course entity to establish the other side of the relationship:

```
import {
  Entity,
  // ...existing code...
  ManyToMany,
  // ...existing code...
} from 'typeorm';
import { Department } from '../../departments/entities/department.entity';
import { Student } from 'src/students/entities/student.entity';

@Entity()
export class Course {
  // ...existing code...

  @ManyToMany(() => Student, (student) => student.courses)
  students: Relation<Student[]>;
}
```

Note that we don't use the `@JoinTable()` decorator on this side, as it's already defined in the Student entity.

3. Update the Student Module

When working with related entities, we need to import them in the module:

```
@Module({
  imports: [
    DatabaseModule,
    TypeOrmModule.forFeature([Student, Profile, Course]),
  ],
  controllers: [StudentsController],
  providers: [StudentsService],
})
export class StudentsModule {}
```

Similarly, in the Course module:

```
@Module({
  imports: [
    DatabaseModule,
    TypeOrmModule.forFeature([Course, Department, Student]),
  ],
  providers: [CoursesService],
  controllers: [CoursesController],
})
export class CoursesModule {}
```

Managing the Many-to-Many Relationship

1. Student Service Methods

To manage enrollments from the student perspective:

```
async enrollStudentInCourse(studentId: number, courseId: number): Promise<Student>
{
  // Find the student with courses relation
  const student = await this.studentRepository.findOne({
    where: { id: studentId },
    relations: ['courses'],
  });

  if (!student) {
    throw new NotFoundException(`Student with ID ${studentId} not found`);
  }

  // Find the course
  const course = await this.courseRepository.findOneBy({ id: courseId });
  if (!course) {
    throw new NotFoundException(`Course with ID ${courseId} not found`);
  }

  // Initialize courses array if it doesn't exist
  if (!student.courses) {
    student.courses = [];
  }

  // Check if already enrolled
  const isAlreadyEnrolled = student.courses.some(
    (enrolledCourse) => enrolledCourse.id === courseId,
  );

  if (!isAlreadyEnrolled) {
    student.courses.push(course);
    await this.studentRepository.save(student);
  }
```

```
    return student;
  }

  async unenrollStudentFromCourse(studentId: number, courseId: number):
  Promise<Student> {
    // Find the student with courses relation
    const student = await this.studentRepository.findOne({
      where: { id: studentId },
      relations: ['courses'],
    });

    if (!student) {
      throw new NotFoundException(`Student with ID ${studentId} not found`);
    }

    // Check if the student is enrolled in the course
    if (!student.courses || student.courses.length === 0) {
      throw new NotFoundException(
        `Student with ID ${studentId} is not enrolled in any courses`,
      );
    }

    // Filter out the course to unenroll from
    student.courses = student.courses.filter(
      (course) => course.id !== courseId,
    );

    // Save the updated student
    return this.studentRepository.save(student);
  }

  async getStudentCourses(studentId: number): Promise<Course[]> {
    const student = await this.studentRepository.findOne({
      where: { id: studentId },
      relations: ['courses'],
    });

    if (!student) {
      throw new NotFoundException(`Student with ID ${studentId} not found`);
    }

    return student.courses || [];
  }

  async updateStudentCourses(studentId: number, courseIds: number[]):
  Promise<Student> {
    // Find the student with courses relation
    const student = await this.studentRepository.findOne({
      where: { id: studentId },
      relations: ['courses'],
    });

    if (!student) {
      throw new NotFoundException(`Student with ID ${studentId} not found`);
    }

    // Find all courses by IDs
    const courses = await this.courseRepository.findBy({
      id: In(courseIds),
    });

    if (courses.length !== courseIds.length) {
      const foundIds = courses.map((course) => course.id);
      const missingIds = courseIds.filter((id) => !foundIds.includes(id));
      throw new NotFoundException(
        `Courses with IDs ${missingIds.join(', ')} not found`,
      );
    }

    // Replace student's courses with the new selection
    student.courses = courses;

    // Save the updated student
    return this.studentRepository.save(student);
  }
```

2. Course Service Methods

Similarly, you can manage enrollments from the course perspective:

```
  async getEnrolledStudents(courseId: number): Promise<Student[]> {
    const course = await this.courseRepository.findOne({
      where: { id: courseId },
      relations: ['students', 'students.profile'], // Include profile information
    });

    if (!course) {
      throw new NotFoundException(`Course with ID ${courseId} not found`);
    }

    return course.students || [];
  }

  async addStudentToCourse(courseId: number, studentId: number): Promise<Course> {
    // Find the course with students relation
    const course = await this.courseRepository.findOne({
      where: { id: courseId },
      relations: ['students'],
    });

    if (!course) {
      throw new NotFoundException(`Course with ID ${courseId} not found`);
    }
    // Find the student
    const student = await this.studentRepository.findOneBy({ id: studentId });
    if (!student) {
      throw new NotFoundException(`Student with ID ${studentId} not found`);
    }

    // Initialize students array if it doesn't exist
    if (!course.students) {
      course.students = [];
    }

    // Check if student is already enrolled
    const isAlreadyEnrolled = course.students.some(
      (enrolledStudent) => enrolledStudent.id === studentId,
    );

    if (!isAlreadyEnrolled) {
      course.students.push(student);
      await this.courseRepository.save(course);
    }

    return course;
  }

  async removeStudentFromCourse(courseId: number, studentId: number):
  Promise<Course> {
    // Find the course with students relation
    const course = await this.courseRepository.findOne({
      where: { id: courseId },
      relations: ['students'],
    });

    if (!course) {
      throw new NotFoundException(`Course with ID ${courseId} not found`);
    }

    // Check if the course has any enrolled students
    if (!course.students || course.students.length === 0) {
      throw new NotFoundException(
        `Course with ID ${courseId} has no enrolled students`,
      );
    }

    // Filter out the student to remove
    course.students = course.students.filter(
      (student) => student.id !== studentId,
    );

    // Save the updated course
    return this.courseRepository.save(course);
  }
```

3. Controller Endpoints

Both controllers need endpoints to manage the relationship:

Student Controller:

```
// http://localhost:8000/students/1/courses
@Get(':id/courses')
getStudentCourses(@Param('id', ParseIntPipe) id: number) {
  return this.studentsService.getStudentCourses(id);
}

// http://localhost:8000/students/1/courses/2
@Post(':studentId/courses/:courseId')
enrollStudentInCourse(
  @Param('studentId', ParseIntPipe) studentId: number,
  @Param('courseId', ParseIntPipe) courseId: number,
) {
  return this.studentsService.enrollStudentInCourse(studentId, courseId);
}

// http://localhost:8000/students/1/courses/2
@Delete(':studentId/courses/:courseId')
unenrollStudentFromCourse(
  @Param('studentId', ParseIntPipe) studentId: number,
  @Param('courseId', ParseIntPipe) courseId: number,
) {
  return this.studentsService.unenrollStudentFromCourse(studentId, courseId);
}

// http://localhost:8000/students/1/courses
@Patch(':id/courses')
updateStudentCourses(
  @Param('id', ParseIntPipe) id: number,
  @Body() courseIds: number[],
) {
  return this.studentsService.updateStudentCourses(id, courseIds);
}
```

Course Controller:

```
// http://localhost:3000/courses/1/students
@Get(':id/students')
getEnrolledStudents(@Param('id', ParseIntPipe) id: number) {
  return this.coursesService.getEnrolledStudents(id);
}

// http://localhost:3000/courses/1/students/2
@Post(':courseId/students/:studentId')
addStudentToCourse(
  @Param('courseId', ParseIntPipe) courseId: number,
  @Param('studentId', ParseIntPipe) studentId: number,
) {
  return this.coursesService.addStudentToCourse(courseId, studentId);
}

// http://localhost:3000/courses/1/students/2
@Delete(':courseId/students/:studentId')
removeStudentFromCourse(
  @Param('courseId', ParseIntPipe) courseId: number,
  @Param('studentId', ParseIntPipe) studentId: number,
) {
  return this.coursesService.removeStudentFromCourse(courseId, studentId);
}
```

Important Notes About Many-to-Many Relationships:

1. **Join Table**: The `@JoinTable()` decorator must be specified on one (and only one) side of the relationship. This decorator creates the join table in the database.

2. **Synchronization**: When working with many-to-many relationships, TypeORM automatically synchronizes both sides of the relationship. For example, if you enroll a student in a course from the student side, the course's students array will also be updated.

3. **Eager Loading**: By default, relationships are not eagerly loaded. You must explicitly include them using the `relations` option in your query or set `eager: true` in the relationship decorator.

4. **Cascade Options**: You can specify cascade options to automatically handle related entities:

   ```
   @ManyToMany(() => Course, (course) => course.students, {
     cascade: true, // or ['insert', 'update', 'remove']
   })
   ```

5. **Database Impact**: Many-to-many relationships require a join table, which can impact performance for large datasets. For very large systems, consider optimizing your queries or using a different relationship pattern.

6. **Bidirectional vs. Unidirectional**: The examples show a bidirectional relationship, but you can also create unidirectional many-to-many relationships by omitting the second parameter in the `@ManyToMany()` decorator.

By following this approach, you create a clean, maintainable codebase for managing complex relationships between students and courses.