Relations Between Entities

TypeORM supports relationships like:

1. **One-to-One (`@OneToOne`)**
2. **One-to-Many (`@OneToMany`)**
3. **Many-to-One (`@ManyToOne`)**
4. **Many-to-Many (`@ManyToMany`)**

One-to-One Relationship (`@OneToOne`)

Scenario: Each **User** has one **Profile** (one-to-one relationship).

Create `User` Entity

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

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

    @Column()
    firstName: string;

    @Column()
    lastName: string;

    @OneToOne(() => Profile, { cascade: true,onDelete:"CASCADE" }) //
Establishing One-to-One relation
    @JoinColumn() // This column will store the foreign key
    profile: Profile;
}
```

Create `Profile` Entity

```
import { Entity, PrimaryGeneratedColumn, Column } from "typeorm";

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

    @Column()
    age: number;

    @Column()
    bio: string;
}
```

Explanation

- `@OneToOne(() => Profile)` → Links **User** to **Profile**.
- `@JoinColumn()` → Stores **foreign key** in the **User table**.
- `{ cascade: true }` → Automatically creates a **Profile** when creating a **User**.

Saving a User with a Profile

```
async function createUserWithProfile() {
    const userRepository = AppDataSource.getRepository(User);

    const user = userRepository.create({

        name: "John",
        email: "xy@gmail.com",
        profile: {
            age: 35,
            designation: "Developer"
        }
    });
    await userRepository.save(user);
    console.log("User with profile saved successfully!");
}

createUserWithProfile();
```

One-to-Many & Many-to-One Relationships (`@OneToMany` & `@ManyToOne`)

Scenario: A **User** can have **multiple Posts** (One-to-Many). Each **Post** belongs to **one User** (Many-to-One).

Create `User` Entity

```
import { Entity, PrimaryGeneratedColumn, Column, OneToMany } from
"typeorm";
import { Post } from "./Post";

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

    @Column()
    firstName: string;

    @Column()
    lastName: string;

    @OneToMany(() => Post, (post) => post.user) // One User → Many Posts
    posts: Post[];
}
```

Create `Post` Entity

```
import { Entity, PrimaryGeneratedColumn, Column, ManyToOne } from
"typeorm";
import { User } from "./User";

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

    @Column()
    title: string;

    @Column("text")
    content: string;

    @ManyToOne(() => User, (user) => user.posts) // Many Posts → One User
    user: User;
}
```

Saving Data

```
async function createPostForUser(userId: number) {
  const userRepository = AppDataSource.getRepository(User);
  const postRepository = AppDataSource.getRepository(Post);

  const user = await userRepository.findOne({ where: { id: userId } });
  if (!user) {
    console.log("User not found");
    return;
  }

  const post = new Post();
  post.title = "My First Post";
  post.content = "This is a sample post";
  post.user = user; // Linking the post to the user

  await postRepository.save(post);
  console.log("Post saved successfully!");
}

createPostForUser(1);
```

Many-to-Many Relationship in TypeORM –

A **Many-to-Many (M:N)** relationship occurs when multiple records in one table are associated with multiple records in another table. Since relational databases don't directly support M:N relationships, we create a **junction table** (also called a bridge table) to connect them.

Example : Students & Courses Enrollment System

Consider an educational system where:
One student can enroll in **multiple courses**
One course can have **multiple students**

This forms a **Many-to-Many** relationship between `Student` and `Course`. To manage this relationship, we introduce an intermediate table called `Enrollment`, which stores:

- `studentId` (reference to `Student`)
- `courseId` (reference to `Course`)
- `enrollmentDate`

Entity Diagram

```
 Student                Enrollment             Course
 ---------     1   M    -------------    M   1  ---------
 id          -------->  studentId      <------  id
 name                   courseId                name
 email                  enrollmentDate          duration
```

1 Create Student Entity

```
import { Entity, PrimaryGeneratedColumn, Column, ManyToMany, JoinTable }
from "typeorm";
import { Course } from "./Course";
```

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

    @Column()
    name: string;

    @Column()
    email: string;

    @ManyToMany(() => Course, (course) => course.students)
    @JoinTable()  // This creates a junction table automatically
    courses: Course[];
}
```

2 Create Course Entity

```
import { Entity, PrimaryGeneratedColumn, Column, ManyToMany } from
"typeorm";
import { Student } from "./Student";

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

    @Column()
    name: string;

    @Column()
    duration: string;

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

Note:

- `@ManyToMany` is used to define the relationship in both `Student` and `Course`.
- `@JoinTable()` is used only in one entity (`Student`), so TypeORM knows this is the **owner side** of the relation.

CRUD Operations for Many-to-Many Relationship

3 Add a Student with Multiple Courses

```
import { AppDataSource } from "./data-source";
import { Student } from "./entities/Student";
import { Course } from "./entities/Course";

const addStudentWithCourses = async () => {
    const studentRepo = AppDataSource.getRepository(Student);
    const courseRepo = AppDataSource.getRepository(Course);

    const course1 = await courseRepo.findOneBy({ id: 1 });
    const course2 = await courseRepo.findOneBy({ id: 2 });

    if (!course1 || !course2) return console.log("Courses not found");

    const newStudent = studentRepo.create({
        name: "John Doe",
        email: "john@example.com",
        courses: [course1, course2],
    });

    await studentRepo.save(newStudent);
    console.log("Student with courses saved successfully!");
};

addStudentWithCourses();
```

4 Fetch Students with Enrolled Courses

```
const getStudentsWithCourses = async () => {
    const studentRepo = AppDataSource.getRepository(Student);
    const students = await studentRepo.find({ relations: ["courses"] });

    console.log(JSON.stringify(students, null, 2));
};

getStudentsWithCourses();
```

5 Enroll an Existing Student in a New Course

```
const enrollStudentInCourse = async (studentId: number, courseId: number)
=> {
    const studentRepo = AppDataSource.getRepository(Student);
    const courseRepo = AppDataSource.getRepository(Course);

    const student = await studentRepo.findOne({ where: { id: studentId },
relations: ["courses"] });
    const course = await courseRepo.findOneBy({ id: courseId });

    if (!student || !course) return console.log("Student or Course not
found");

    student.courses.push(course);
    await studentRepo.save(student);

    console.log("Student enrolled in the new course successfully!");
};

enrollStudentInCourse(1, 3);
```

key points

Many-to-Many relationships require a **junction table**.

`@ManyToMany` defines the relation, and `@JoinTable()` specifies the owner side.
CRUD operations allow adding, fetching, and updating related data.