Introduction to TypeORM

What is TypeORM?

TypeORM is an **Object-Relational Mapper (ORM)** for **TypeScript and JavaScript**, designed to work with databases like **SQL Server, MySQL, PostgreSQL, and SQLite**. It allows us to interact with the database using **TypeScript classes and decorators** instead of writing raw SQL queries.

Why Use TypeORM?

Object-Oriented Approach – Works with TypeScript classes.
Simplifies Database Operations – No need to write complex SQL queries.
Supports Migrations – Helps version-control database schema.
Cross-Database Compatibility – Works with different databases (SQL Server, MySQL, PostgreSQL, etc.).
Built-in Query Builder – Allows flexible query construction.

Setting Up a TypeORM Project

Step 1: Create a New TypeORM Project

Step 2: Install Required Dependencies

npm install typeorm@0.3.17 reflect-metadata@0.1.13 mssql@9.1.1 dotenv@16.3.1

npm install --save-dev typescript@5.2.2 ts-node@10.9.1 @types/node@20.5.1

Step 3: Configure TypeORM Data Source (SQL Server Connection)

Example

TypeScript Configuration

In `tsconfig.json`, ensure the following settings:

```
{
  "compilerOptions": {
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
     "outDir": "./disc",
     "rootDir": "./src",

    "strictPropertyInitialization": false,
```

```
    "target": "ES6",
    "moduleResolution": "Node"
  }
}
```

Step 1: Create a `.env` File for Database Configuration

Create a **.env** file in the root directory and add your **SQL Server credentials**:

```
DB_HOST=localhost
DB_PORT=1433
DB_USER=sa
DB_PASSWORD=yourStrong Password
DB_NAME=TestDB
```

Step 2: Create a datasource File

```
import "reflect-metadata";
import { DataSource } from "typeorm";
import * as dotenv from "dotenv";

dotenv.config();

export const AppDataSource = new DataSource({
    type: "mssql",  // Using SQL Server
    host: process.env.DB_HOST || "localhost",
    port: Number(process.env.DB_PORT) || 1433,
    username: process.env.DB_USER || "sa",
    password: process.env.DB_PASSWORD || "yourStrong(!)Password",
    database: process.env.DB_NAME || "TestDB",
    synchronize: true,  // Auto-create tables (for development)
    logging: true,
    entities: ["src/entities/*.ts"],  // Path to entity files
    options: {
        encrypt: false, // Disable SSL for local development
        enableArithAbort: true
    }
});
```

Step 3: Create an Entity

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


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

```
    @Column({ type: "varchar", length: 255 })
    firstName: string;

    @Column({ type: "varchar", length: 255 })
    lastName: string;

    @Column({ type: "varchar", length: 255 ,unique: true })
    email: string;

    @Column({ type: "bit", default: () => "1" })
    isActive: boolean;
}
```

Step 4: Initialize the Database Connection

`src/index.ts` to **connect to the database** and perform basic operations.

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

AppDataSource.initialize()
    .then(async () => {
        console.log(" Database connected successfully!");

        // Create a new user instance
        const userRepository = AppDataSource.getRepository(User);

        const newUser = userRepository.create({
            firstName: "John",
            lastName: "Doe",
            email: "john.doe@example.com",
        });
```

```
    // Save user to the database
    await userRepository.save(newUser);
    console.log(" New user saved:", newUser);

    // Fetch all users
    const users = await userRepository.find();
    console.log(" All Users:", users);
   //find user by id
   const user = await userRepository.findOne({ where: { id:3} });
  //find user by email
  const userByEmail = await userRepository.findOneBy({ email:"st@gmail.com"
});

//update record
await userRepository.update(userId, { firstName: newFirstName });
console.log("User updated successfully!");

//delete record
await userRepository.delete(userId); console.log("User deleted successfully!");

  })
  .catch((error) => console.log(" Error connecting to the database:", error));
```

Step 5: Run the Application

Start the TypeScript project using `ts-node`:

```
npx ts-node src/index.ts
```

Entity Decorators

Decorator	Description
`@Entity()`	Marks the class as an Entity (database table)
`@PrimaryGeneratedColumn()`	Defines a **Primary Key** (auto-incremented)
`@Column()`	Defines a **column** in the table
`@Column({ unique: true })`	Adds a **unique constraint**
`@Column({ default: value })`	Sets a **default value**

Column Data Types

TypeORM supports different **column types** based on the database.

String Column

```
@Column({ type: "varchar", length: 255 })
name: string;
```

Assignment (Day 1)

Each assignment requires:
Setting up a TypeORM project
Creating an entity with basic fields
Implementing CRUD operations using Express

1 Employee Management System

- **Entity**: `Employee`
- **Fields**: `id`, `name`, `email`, `position`, `salary`
- **Operations**: CRUD for employees
- **Use Case**: Manage employees in a company

2 Student Management System

- **Entity**: `Student`
- **Fields**: `id`, `name`, `age`, `email`, `course`
- **Operations**: CRUD for students
- **Use Case**: Manage students in an institute

3 Book Management System

- **Entity**: `Book`
- **Fields**: `id`, `title`, `author`, `publishedYear`, `price`
- **Operations**: CRUD for books
- **Use Case**: Manage books in a library

4 Product Inventory System

- **Entity**: `Product`
- **Fields**: `id`, `name`, `category`, `price`, `stock`
- **Operations**: CRUD for products
- **Use Case**: Manage products in a warehouse

5 Customer Management System

- **Entity**: `Customer`
- **Fields**: `id`, `name`, `email`, `phone`, `address`
- **Operations**: CRUD for customers
- **Use Case**: Store customer details in a shopping app

6 Task Management System

- **Entity**: `Task`
- **Fields**: `id`, `title`, `description`, `status`, `dueDate`
- **Operations**: CRUD for tasks
- **Use Case**: Manage to-do lists

7 Movie Database System

- **Entity**: `Movie`
- **Fields**: `id`, `title`, `genre`, `releaseYear`, `rating`
- **Operations**: CRUD for movies
- **Use Case**: Store and retrieve movie details

8 Car Rental System

- **Entity**: `Car`
- **Fields**: `id`, `brand`, `model`, `year`, `rentalPrice`
- **Operations**: CRUD for cars
- **Use Case**: Manage rental cars

9 Hotel Room Booking System

- **Entity**: `Room`
- **Fields**: `id`, `roomNumber`, `type`, `price`, `availability`
- **Operations**: CRUD for rooms
- **Use Case**: Track hotel room bookings

10 Online Shopping System

- **Entity**: `Order`
- **Fields**: `id`, `orderNumber`, `customerName`, `totalAmount`, `status`
- **Operations**: CRUD for orders
- **Use Case**: Track customer orders

11 Job Listing System

- **Entity**: `Job`
- **Fields**: `id`, `title`, `company`, `location`, `salary`
- **Operations**: CRUD for job postings
- **Use Case**: Manage job listings for a job portal

12 Gym Membership System

- **Entity**: `Member`
- **Fields**: `id`, `name`, `email`, `membershipType`, `joiningDate`
- **Operations**: CRUD for members
- **Use Case**: Track gym members

13 Event Management System

- **Entity**: `Event`
- **Fields**: `id`, `name`, `date`, `location`, `organizer`
- **Operations**: CRUD for events
- **Use Case**: Organize and track events

14 Medical Appointment System

- **Entity**: `Appointment`
- **Fields**: `id`, `patientName`, `doctorName`, `appointmentDate`, `status`
- **Operations**: CRUD for appointments
- **Use Case**: Schedule and manage doctor appointments

15 Feedback Collection System

- **Entity**: `Feedback`
- **Fields**: `id`, `userName`, `email`, `message`, `rating`
- **Operations**: CRUD for feedback
- **Use Case**: Collect and store user feedback

16 Recipe Management System

- **Entity**: `Recipe`
- **Fields**: `id`, `name`, `ingredients`, `instructions`, `cookingTime`
- **Operations**: CRUD for recipes
- **Use Case**: Store and retrieve cooking recipes

17 Music Playlist System

- **Entity**: `Song`
- **Fields**: `id`, `title`, `artist`, `album`, `duration`
- **Operations**: CRUD for songs
- **Use Case**: Create and manage music playlists

18 Complaint Management System

- **Entity**: `Complaint`
- **Fields**: `id`, `userName`, `description`, `status`, `submittedAt`
- **Operations**: CRUD for complaints
- **Use Case**: Track complaints in a service center

19 Weather Report System

- **Entity**: `WeatherReport`
- **Fields**: `id`, `city`, `temperature`, `humidity`, `condition`
- **Operations**: CRUD for weather reports
- **Use Case**: Store and access weather data