- **`What is TypeORM`**

TypeORM is an Object-Relational Mapper (ORM) for TypeScript and JavaScript (ES7+), which allows developers to interact with databases using a more object-oriented approach. It supports various databases, such as MySQL, PostgreSQL, SQLite, MariaDB, and others. TypeORM simplifies database queries, migrations, and data handling by letting you work with entities and decorators, making it popular in the Node.js ecosystem, especially with NestJS.

Key Features of TypeORM:

- **Supports various databases** (MySQL, PostgreSQL, SQLite, MariaDB, SQL Server, etc.)
- **Database migrations** to manage schema changes
- **Repository pattern** for database interactions
- **TypeScript support** with strong typing
- **Entity relationships** (one-to-one, one-to-many, many-to-many)
- **Eager and lazy loading** of related data
- **Query builder** for advanced queries

- **`upsert()` Method:**
  - a notable method introduced is **`upsert()`**, which simplifies the process of updating existing records or creating new ones if they don't exist. This method is particularly useful for handling conflicts in unique constraints (like emails or usernames).
  - **Example of upsert() Method:**

```
const userRepository = myDataSource.getRepository(User);
await userRepository.upsert(
    [
      { email: 'hello@example.com', name: 'Updated Name' },
```

```
      { email: 'goodbye@example.com', name: 'New User' }
    ],
    ['email']
);
```

- The `upsert()` method handles conflict resolution by checking the `email` field and updating it if a match is found. If no match exists, a new entry is created.

- **Get FullName**

  - **Example of get full name:**

```
async getFullNames(): Promise<{ fullName: string }[]> {
    return await this.userRepository
      .createQueryBuilder('user')
      .select("CONCAT(user.FirstName, ' ', user.LastName)", 'fullName')
      .getRawMany(); // Get raw results as an array of objects with
fullName field
  }
```

  - **Output Example:**

```
Data: [
  { "fullName": "Ayush Donga" },
  { "fullName": "John Doe" }
]
```

- `find` **Method**

- The `find` method retrieves multiple records.

- **Example of find:**

```
const userRepository = myDataSource.getRepository(User);
 // Find all users
 const users = await userRepository.find();
 console.log('All Users:', users);

// Find users with specific criteria
const activeUsers = await userRepository.find({ where: { isActive:
true } });
console.log('Active Users:', activeUsers);
```

- **Output Example:**

```
  allUsers: [
    { id: 1, name: 'Ayush Donga', isActive: true },
    { id: 2, name: 'Jane Smith', isActive: false }
  ]

  activeUsers: [
    { id: 1, name: 'Ayush Donga', isActive: true }
  ]
```

- **`findOne` Method**
  - The findOne method retrieves a single record.

  - **Example of find:**

```
async function findUserById(userId: number) {
```

```
    const userRepository = myDataSource.getRepository(User);

    // Find one user by ID
    const user = await userRepository.findOne({ where: { id: userId
} });
    console.log('User by ID:', user);
  }

  findUserById(1);
```

- **Output Example:**

```
userByID: { id: 1, name: 'John Doe', isActive: true }
```

- **Example of user creating and updating:**

  - **Example of create or update user:**

```
// create-user.dto.ts
export class CreateUserDto {
  name: string;
  email: string;
  password: string;
}
```

```
//update-user.dto-ts
export class UpdateUserDto {
  name?: string;
  email?: string;
  password?: string;
  isActive?: boolean;
}
```

```ts
// user.service.ts
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { User } from './user.entity';
import { CreateUserDto } from './dto/create-user.dto';
import { UpdateUserDto } from './dto/update-user.dto';

@Injectable()
export class UserService {
  constructor(
    @InjectRepository(User)
    private readonly userRepository: Repository<User>,
  ) {}

  async create(createUserDto: CreateUserDto): Promise<User> {
    const user = this.userRepository.create(createUserDto);
    return this.userRepository.save(user);
  }

  async update(id: number, updateUserDto: UpdateUserDto):
Promise<User> {
    await this.userRepository.update(id, updateUserDto);
    return this.userRepository.findOne({ where: { id } });
  }
}
```

- **Output Example:**

```
createdUser : {
  "id": 1,
  "name": "John Doe",
  "email": "johndoe@example.com",
  "isActive": true
}

updatedUser : {
  "id": 1,
  "name": "John Updated",
  "email": "johnupdated@example.com",
  "isActive": true
}
```

- `delete` **Method:**

  - **Example of delete user:**

```
async function deleteUser(id: number) {
    const userRepository = getRepository(User);

    // Delete the user with the specified ID
    const result = await userRepository.delete(id);

    console.log("Delete Result:", result);
}
```

  - **Output Example:**

```
 Result: {
    raw: {
        affectedRows: 1  // Example: This may vary depending on
the database used
    },
    affected: 1  // Number of rows that were deleted
}
```

- **One-to-One Relationship Example**

  - **Example of one to one:**

```
const profile = new Profile();
profile.bio = "Software Engineer";

const user = new User();
user.name = "John Doe";
user.profile = profile;

await userRepository.save(user);

const savedUser = await userRepository.findOne({
  where: { id: 1 },
  relations: ["profile"],
});
console.log(savedUser);
```

  - **Output Example:**

```
{
   "id": 1,
   "name": "John Doe",
   "profile": {
     "id": 1,
     "bio": "Software Engineer"
   }
}
```

- **One-to-Many Relationship Example**

  - **Example of one to many:**

```
import { getRepository } from 'typeorm';
import { User } from './entities/User';
import { Order } from './entities/Order';
import { Product } from './entities/Product';
import { Category } from './entities/Category';
import { Supplier } from './entities/Supplier';

async function getUserOrderDetails() {
  const userRepository = getRepository(User);

  const result = await userRepository
    .createQueryBuilder('user')
    .leftJoinAndSelect('user.orders', 'order')
    .leftJoinAndSelect('order.products', 'product')
    .leftJoinAndSelect('product.categories', 'category')
    .leftJoinAndSelect('category.suppliers', 'supplier')
```

```
    .select([
      'user.id',
      'user.name',
      'order.id',
      'order.userId',
      'product.id',
      'product.name',
      'category.id',
      'category.name',
      'supplier.id',
      'supplier.name',
    ])
    .getMany();

  return result;
}
```

- **Output Example:**

```
data: [
    {
      "id": 1,
      "name": "John Doe",
      "orders": [
        {
          "id": 1,
          "userId": 1,
          "products": [
            {
              "id": 1,
              "name": "Product 1",
```

```
            "categories": [
              {
                "id": 1,
                "name": "Category 1",
                "suppliers": [
                  {
                    "id": 1,
                    "name": "Supplier 1"
                  },
                  {
                    "id": 2,
                    "name": "Supplier 2"
                  }
                ]
              }
            ]
          }
        ]
      }
    ]
  },
  {
    "id": 2,
    "name": "Jane Doe",
    "orders": [
      {
        "id": 2,
        "userId": 2,
        "products": [
          {
```

```
            "id": 2,
            "name": "Product 2",
            "categories": [
              {
                "id": 2,
                "name": "Category 2",
                "suppliers": [
                  {
                    "id": 3,
                    "name": "Supplier 3"
                  }
                ]
              }
            ]
          }
        ]
      }
    ]
  }
]
```

Table 1: `users`

id	name	email
1	John Doe	john@example.com
2	Jane Doe	jane@example.com

Table 2: `orders`

id	user_id	order_date
1	1	2024-09-01
2	2	2024-09-02

Table 3: `products`

id	name	order_id	price
1	Product 1	1	50
2	Product 2	1	30
3	Product 3	2	60

Table 4: `categories`

id	product_id	name
1	1	Electronics
2	2	Home Appliances
3	3	Furniture

Table 5: `suppliers`

id	category_id	name
1	1	Supplier 1
2	2	Supplier 2
3	3	Supplier 3

Relationships

- `users` → `orders`: One-to-many (each user can have multiple orders).
- `orders` → `products`: One-to-many (each order can have multiple products).
- `products` → `categories`: One-to-many (each product belongs to one category).
- `categories` → `suppliers`: One-to-many (each category can have multiple suppliers).