**ntro to Typeorm**

TypeORM is an [ORM](https://en.wikipedia.org/wiki/Object-relational_mapping) that can run in NodeJS, Browser, Cordova, PhoneGap, Ionic, React Native, NativeScript, Expo, and Electron platforms and can be used with TypeScript and JavaScript (ES5, ES6, ES7, ES8)

from typeorm.io

**Getting Started**

Typeorm is like any other ORM it allows to interact with multiple databases and supports multiple database drivers for typeorm it supports MySQL, MariaDB, PostgreSQL, CockroachDB, SQLite, Microsoft SQL Server sql.js Oracle and MongoDB for NoSQL Databases so depending on your case you just simply need to install the corresponding driver of your database and you are ready to go.

Typeorm comes with a robust CLI that allows us to manage our projects and interact with the database with an easy to use CLI.

npm install typeorm -g

Make sure to install the CLI globally.

Now let’s create and initialize a new Project

typeorm init --name Typeorm-project --database mysql

You must explicitly specify your database so the proper driver will be installed for you on the project.

Now, simply cd to your project and open it with you, favourite code editor. I’m using VSCode.

$ cd typeorm-project && code .

**Typeorm Configuration**

We need to provide our info and custom configuration depending on our project to the typeorm to be able to connect and interact with the database properly.

So a file named ormconfig.json will be created on the root of the project that has all the configuration that typeorm needs.

{  
 "type": "mysql",  
 "host": "localhost",  
 "port": 3306,  
 "username": "root",  
 "password": "",  
 "database": "social\_network",  
 "synchronize": true,  
 "logging": false,  
 "migrationsTableName": "migrations",  
 "entities": ["src/entity/\*\*/\*.ts"],  
 "migrations": ["src/migration/\*\*/\*.ts"],  
 "subscribers": ["src/subscriber/\*\*/\*.ts"],  
 "cli": {  
 "entitiesDir": "src/entity",  
 "migrationsDir": "src/migration",  
 "subscribersDir": "src/subscriber"  
 }  
}

Most of the configuration is already set for you by the CLI you only need to change your database username and password to allow typeorm to connect to the database and make sure to create a database and provide its name on the config.

the migrations, entities and subscribers directories have to be provided otherwise the underlining feature won’t properly work since this allows typeorm core to identify the entities and then exposes the entities instances through the provided API.

**Typeorm Entities**

Entity is a class that represents a database table the filed of the class are the columns of the table, typeorm uses the help of decorators to mark the class as an entity and maps it to the database as a table with the provided fields.

Every entity has to be marked with @Entity decorator to tell typeorm that it is an entity.

The CLI creates a User entity for us

import { Entity, PrimaryGeneratedColumn, Column, OneToMany } from "typeorm";  
import { Tweet } from "./Tweet";  
  
@Entity({ name: "users" })  
export class User {  
 @PrimaryGeneratedColumn("uuid")  
 id: string;  
  
 @Column()  
 firstName: string;  
  
 @Column()  
 lastName: string;  
  
 @Column({ type: "varchar" })  
 age: number;  
}

For every field it has to be marked using @Column decorator, the type of the column will be abstracted from the attribute type using the help of typescript or you can provide a custom type through the column decorator.

You can also specify the length of the column and if nullable or unique.

Each entity will be automatically added to the typeorm connection object since it resides in the entity folder specified in the ormconfig.json file.

Let’s create another Entity and since we got a social network example lets make a tweet entity that allows the user to post tweets.

import { Entity, PrimaryGeneratedColumn, Column, ManyToOne } from "typeorm";  
import { User } from "./User";  
  
@Entity({ name: "tweets" })  
export class Tweet {  
 @PrimaryGeneratedColumn("uuid")  
 id: string;  
  
 @Column({ type: "varchar", length: 80 })  
 title: string;  
  
 @Column({ type: "varchar", length: 300 })  
 content: string;  
}

For the id column, we are using special decorator PrimaryGeneratedColumn which auto-generates the id for us and we can provide it either to use default incremental ids (ex. 1 2 3…) or generate a UUID for every tweet which is better for security reasons.

**Creating and Saving Instances**

Typeorm has two main patterns to work with entity instances (Entity Manager and Entity Repository) the CLI generates an example using Entity Manager so will be using Entity Repository pattern to better understand how it works.

import { getRepository } from "typeorm";  
import { User } from "./entity/User";  
  
export const Bootstrap = async () => {  
 const userRepo = getRepository(User);  
 const user = userRepo.create({  
 firstName: "Alex",  
 lastName: "Brooks",  
 age: 22,  
 });  
 await userRepo.save(user).catch((err) => {  
 console.log("Error: ", err);  
 });  
 console.log("New User Saved", user);  
};

Create a boostrap.js file in the root directory which going to take care of creating and storing a new instance of the user.

For Entity Repository pattern we always need to get the entity repository using getRepository function and providing it with the Entity class we wish to get.

We simply create an instance with user data and afterwards for storing it into the database we call repo.save method with the user instance to save, the save method will return a promise so it is better to use async/await to await for the promise to be fulfilled and don’t forget to define your catch method if any errors occur.

The API is pretty simple to create and save the new instance to the database.

**Typeorm Entity Relations**

Relationships between entities is a very important aspect to cover for every ORM since it allows multiple entities to have relations between each other to form a specific structure for the developer’s needs.

Typeorm Relations are defined in the underlined entities’ classes using the help of a decorator.

So, let’s create a relation between a user and tweet where each user can have many tweets but a tweet belongs to a single user.

import { Entity, PrimaryGeneratedColumn, Column, OneToMany } from "typeorm";  
import { Tweet } from "./Tweet";  
  
@Entity({ name: "users" })  
export class User {  
 @PrimaryGeneratedColumn("uuid")  
 id: string;  
  
 @Column()  
 firstName: string;  
  
 @Column()  
 lastName: string;  
  
 @Column()  
 age: number;  
  
 @OneToMany((type) => Tweet, (tweet) => tweet.user)  
 tweets: Tweet[];  
}

We use oneToMany relation decorator to tell that user can have many tweets.

The decorator takes the type of the second part of the relation, in this case, it is the Tweet entity and the second type is the attribute we can access through the user from the Tweet entity so typeorm will use this to allow two-way relation accessing even though the foreign keys are not on that entity.

We also need to put the relationship on the tweet entity.

import { Entity, PrimaryGeneratedColumn, Column, ManyToOne } from "typeorm";  
import { User } from "./User";  
  
@Entity({ name: "tweets" })  
export class Tweet {  
 @PrimaryGeneratedColumn("uuid")  
 id: string;  
  
 @Column({ type: "varchar", length: 80 })  
 title: string;  
  
 @Column({ type: "varchar", length: 300 })  
 content: string;  
  
 @ManyToOne((type) => User, (user) => user.tweets)  
 user: User;  
}

Since a tweet can belong only to one user so we use ManyToOne decorator to describe the relationship.

Now use has many tweets and a tweet belongs to a single user.

Let’s make the user tweet a new tweet.

export const Bootstrap = async () => {  
 const userRepo = getRepository(User);  
 const user = userRepo.create({  
 firstName: "Alex",  
 lastName: "Brooks",  
 age: 22,  
 });  
 await userRepo.save(user).catch((err) => {  
 console.log("Error: ", err);  
 });  
 console.log("New User Saved", user);  
  
 const tweetRepo = getRepository(Tweet);  
 const tweet = new Tweet();  
 tweet.title = "I finally got a new Job!";  
 tweet.content = "Well after a long time I landed my dream job on Netflix";  
 tweet.user = user;  
  
 await tweetRepo.save(tweet).catch((err) => console.log(err));  
};

In the bootstrap function we create a new user then we create a new tweet for the user and we assign the tweet to the user, ManyToOne or OneToMany relation the foreign key is stored on the Entity that has ManyToOne decorator since it has a single instance of the other type of the entity, in this case, it is the Tweet table that has userId column.

We create a new instance of Tweet entity then we assign the tweet to the user.

After saving the tweet instance the userId column will hold the user id that means the tweet does belong to the user.

**Querying Database**

Typeorm provides a simple API to query and fetch data from the database it has find method or query build which allows building regular and powerful SQL queries with a constructive API.

Let’s create a find function alongside bootstrap function to query some user data.

export const find = async () => {  
 const userRepo = getRepository(User);  
  
 const user = await userRepo  
 .findOne({ where: { firstName: "Alex" } })  
 .catch((err) => {  
 console.log(err);  
 });  
  
 if (user) console.log("User: ", user);  
};

We can simply use find or findOne with where close to explicitly find firstname that matches “Alex”.

It returns a Promise of User or User[] array depending on the method you used to query.

Find methods are eager-loaded which means by default they fetch Entity’s data alongside its related entities data which in some cases it just adds more overhead rather than being useful.

There are two ways you can get over eager loading and auto-loaded relations:

1- Use LazyLoad pattern which allows us to load the relations only when we explicitly ask for it rather than being auto-loaded on the fly.

To make an entity use lazyLoading you have to change the type of its relations to use promise instead of bare types.

@Entity({ name: "users" })  
export class User {  
 @PrimaryGeneratedColumn("uuid")  
 id: string;  
  
 @Column()  
 firstName: string;  
  
 @Column()  
 lastName: string;  
  
 @Column()  
 age: number;  
  
 @OneToMany((type) => Tweet, (tweet) => tweet.user)  
 tweets: Promise<Tweet[]>;  
}

For the user to change the type of tweets from Tweet[] to Promise<Tweet[]>, in this case, typeorm will lazy load the tweets only when you ask for it disabling eager loading.

Now to load the tweets data from a user instance we simply need to await for the promise to be resolved.

export const find = async () => {  
 const userRepo = getRepository(User);  
  
 const user = await userRepo  
 .findOne({ where: { firstName: "Alex" } })  
 .catch((err) => {  
 console.log(err);  
 });  
  
 if (user) console.log("Tweets", await user.tweets);  
};

2- Use QueryBuilder API to query data, by default it won’t fetch relations data until you explicitly tell it to.

So without inner joining, user tweets would be undefined.

import {getRepository} from "typeorm";  
  
const user = await getRepository(User)  
 .createQueryBuilder("user")  
 .where("user.firstName = :firstName", { firstName: "Alex" })  
 .getOne();  
  
console.log("Tweets: ", user.tweets); ///< undefined

You can load tweets with Inner Join

import {getRepository} from "typeorm";  
  
const user = await getRepository(User)  
 .createQueryBuilder("user")  
 .innerJoin("user.tweets", "tweet"})  
 .where("user.firstName = :firstName", { firstName: "Alex" })  
 .getOne();  
  
console.log("Tweets: ", user.tweets); ///< User Tweets

I would rather user lazy Loading pattern with find cause is much simpler rather than the query builder since query builder is designed for complicated and nested queries.

**Typeorm Migrations**

Migrations are a way to alter the structure of your tables without affecting the data or risk losing it, most needed in a production environment where the developer did changes to the entities code base and need to make those changes to the database tables and that’s through migrations.

Usually, a single migration file respects SRP (Single Responsibility Principle) so it does one thing only (forex add a column to user table) this way if you would like to revert the migration it would exactly remove the changes you want.

Migrations can be created through the Typeorm CLI.

typeorm migration:create -n add-role-column-to-users

This will generate a timestamp plus the name provided filename of the migration.

import {  
 MigrationInterface,  
 QueryRunner,  
 TableCheck,  
 TableColumn,  
} from "typeorm";  
  
export class addRoleColumnToUsers1588553161447 implements MigrationInterface {  
 public async up(queryRunner: QueryRunner): Promise<any> {  
 const usersTable = await queryRunner.getTable("users");  
 const roleColumn = new TableColumn({ name: "role", type: "int" });  
 await queryRunner.addColumn(usersTable, roleColumn);  
 }  
  
 public async down(queryRunner: QueryRunner): Promise<any> {  
 const usersTable = await queryRunner.getTable("users");  
 await queryRunner.dropColumn(usersTable, "role");  
 }  
}

The up method is responsible for committing the migration while the down method is responsible for removing it.

The QueryInterface is used to manipulate the tables and it has everything you would need to add a column or alter something in your tables.

In our case, we want to add a role column so we get the users’ table instance with its name and we create new TableColumn instance which takes the name and other column parameters.

Eventually, we add the column to the users’ table and that’s it the role column would be added after running the migration.

typeorm migration:run

The run command will run all pending migrations.

Migrations that had already been run would not run again since typeorm keeps track of the committed migrations in a table in your database, this way it knows exactly what migrations are pending.