CWEB280 -wk8 – LO5 – Db Objects

# TypeORM Entities

TypeORM like most Object Relation Mapping framework use entities classes

## Entities (some other ORMs refer to these classes as Models)

* A class that maps to a database table
* Each property/attribute maps to a column in the database table
* Each instance of the class represents a single row in the database table

Learn more about TypeORM Entities - <https://typeorm.io/#/entities/what-is-entity>

## Decorations

Type ORM uses decorations in the entity class to add metadata (add code not visible to the developer) about the entity and the entity columns.

Decorations are made possible by the reflect-metadata package, already installed your lo4serverapi project.

Example : The decorations are in green and begin with @  
**import** {Entity, PrimaryColumn} **from** "typeorm";

@Entity()

**export** **class** User {

@PrimaryColumn()

id: **number**;

@Column({

type: "varchar",

length: 150,

unique: true,

// ...

})

name: string;

}

## Entity columns - <https://typeorm.io/#/entities/entity-columns>

Since database tables consist of columns entities must contain columns too. Each entity class property you marked with @Column will be mapped to a database table column.

## Primary columns

Each entity must have at least one primary column. There are several types of primary columns, but we will use:

@PrimaryGeneratedColumn() which creates a primary column which value will be automatically generated with an auto-increment value.

## Column options - <https://typeorm.io/#/entities/column-options>

Column options defines additional options for your entity columns. The options include:

Some available options in ColumnOptions:

* type: ColumnType - Column type. Examples (nvarchar, varchar, character, integer, blob, text)
* length: number - Column type's length. Example varchar(150), length would be 50
* width: number - column type's display width. Used only for MySQL integer types
* nullable: boolean - Makes column NULL or NOT NULL in the database.
* select: boolean - Defines whether or not to hide this column by default when making queries.
* default: string - Adds database-level column's DEFAULT value.
* primary: boolean - Marks column as primary. Same if you use @PrimaryColumn.
* unique: boolean - Marks column as unique column (creates unique constraint).

## Class Validator - <https://www.npmjs.com/package/class-validator>

TypeORM does not validate the data stored in the database. Developers must validate the data before it gets saved to the database. The class-validator package uses decorations (like the TypeORM decorations) to specify constraints on the entity properties.

Example:

import {validate, IsInt, Length, IsEmail, Min, Max,} from 'class-validator';

export class Post {

@Length(10, 20,{message: 'Title must between 10 and 20 characters'})

title: string;

@IsInt({message: 'Rating must be a whole number'})

@Min(0,{message: 'Rating must be at least 0'})

@Max(10,{message: 'Rating must be at most 10'})

rating: number;

@IsEmail({},{message: 'Email must be in a valid format'})

email: string;

}

### Validating constraints

Adding the decorations adds meta-data but does not actually enforce the constraints. You need to actually run the validate() or validateOrReject() functions to enforce the contraints

Examples

import {validate} from 'class-validator';

export class PostController {

async function validateOrRejectExample(input) {

try {

await validateOrReject(input);

} catch (errors) {

console.log('Caught promise rejection (validation failed). Errors: ', errors);

}

}

async function validateExample (input) {

const violations = await validate(input);

} if (violations.length) {

console.log(violations);

}

}

}

### Validation errors

The validate method returns an array of ValidationError objects. Each ValidationError has the following properties/attributes

{  
 target: Object; // Object that was validated.

property: string; // Object's property that haven't pass validation.

value: any; // Value that haven't pass a validation.  
 constraints?: { // Constraints that failed validation with error messages.  
 [type: string]: string;  
 };

children?: ValidationError[]; // Contains all nested validation errors of the property  
}

Available Validation Decorators: <https://www.npmjs.com/package/class-validator#manual-validation>

# Add Decorations to User.ts

First install the class-validator package. In the terminal run

npm i class-validator

\src\entity\User.ts: add the code in violet

import {Column, Entity, PrimaryGeneratedColumn} from 'typeorm';  
import {IsOptional, Length, IsInt, Min, Max, MaxLength, IsNotEmpty} from 'class-validator';  
  
@Entity()  
export class User {  
 @PrimaryGeneratedColumn()  
 @IsOptional()  
 id: number;  
  
 @Column('nvarchar', {length: 50})  
 @Length(1, 50, {message: 'First Name must be from $constraint1 to $constraint2 characters '})  
 @IsNotEmpty({message: 'First Name is Required'})  
 firstName: string;  
  
 @Column('nvarchar', {length: 50, nullable: true})  
 @MaxLength( 50, {message: 'Last Name must be at most $constraint1 characters '})  
 @IsOptional()  
 lastName: string;  
  
 @Column('integer')  
 age: number;  
}

\src\controller\UserControler.ts: add import and change the save action/method

import {validate} from 'class-validator';

.

.

.

async save(request: Request, response: Response, next: NextFunction) {  
 // get the metadata/decorations from the User Object and fill with the values in the request body (which does not have any decorations)  
 const newUser = ***Object***.assign(new User(), request.body);

const violations = await validate(newUser, this.validOptions);  
 if (violations.length) {  
 response.statusCode = 422; // Uncrossable Entity  
 return violations;  
 } else {  
 response.statusCode = 201; // Created  
 return this.userRepository.save(newUser);  
 }  
}

Exercise: Delete the sqlite database and restart the app? Did the database get recreated and what are the columns types now?

Exercise: Use Postman to POST (create) a new user: Does it work?   
    {

        "firstName": "Shakira",

        "age": 49

    }

Exercise: Use Postman to POST (create) a new user: Does it work? What does the error look like?  
    {

        "lastName": "Montoya",

        "age": 88

    }

# Make Decorators to replace route.ts

Currently the lo4serverapi project uses route.ts to perform the routing. Using an array of objects to do the routing is cumbersome. We are going add our own code to create the @Controller and @Route decorations.

First create a folder \src\decorator and add the following files

\src\decorator\RouteDefinition.ts

export interface RouteDefinition {  
 // Path to our route  
 param: string;  
 // HTTP Request method (get, post, ...)  
 method: string; // 'get' | 'post' | 'delete' | 'options' | 'put';  
 // Method name within our class responsible for this route  
 action: string;  
}

\src\decorator\Controller.ts

export const Controller = (path = ''): ClassDecorator => {  
 return (target: any) => {  
 Reflect.defineMetadata('path', path, target);  
  
 // Since routes are set by our methods this should almost never be true (except the controller has no methods)  
 if (! Reflect.hasMetadata('routes', target)) {  
 Reflect.defineMetadata('routes', [], target);  
 }  
 };  
};

\src\decorator\Route.ts

import {RouteDefinition} from './RouteDefinition';  
  
export const Route = ( method = 'get', param=''): MethodDecorator => {  
 // `target` equals our class, `propertyKey` equals our decorated method name  
 return (target, propertyKey: string): void => {  
 // In case this is the first route to be registered the `routes` metadata is likely to be undefined at this point.  
 // To prevent any further validation simply set it to an empty array here.  
 if (! Reflect.hasMetadata('routes', target.constructor)) {  
 Reflect.defineMetadata('routes', [], target.constructor);  
 }  
  
 // Get the routes stored so far, extend it by the new route and re-set the metadata.  
 const routes = Reflect.getMetadata('routes', target.constructor) as Array<RouteDefinition>;  
  
 routes.push({  
 method,  
 param,  
 action: propertyKey,  
 });  
 Reflect.defineMetadata('routes', routes, target.constructor);  
 };  
};

Next we need to replace the routing code in index.ts with some new code that will use the decorations for routing

\src\index.ts –the code in white is the old code replaced by the code in violet (the new code)

import UserController from './controller/UserController';  
import {RouteDefinition} from './decorator/RouteDefinition';

.

.

.

// register express routes from defined application routes  
 /\*  
Routes.forEach((route) => {  
 (app as any)[route.method](  
 route.route, ( req: express.Request, res: express.Response, next: express.NextFunction ) => {  
 const result = new (route.controller as any)()[route.action]( req, res, next );  
 if (result instanceof Promise) {  
 result.then((result) => result !== null && result !== undefined ? res.send(result) : next() )  
 .catch((err) => next(createError(500, err)) );  
 } else if (result !== null && result !== undefined) res.json(result);  
 },  
 );  
});  
\*/  
  
 // Iterate over all our controllers and register our routes  
 [  
 UserController,  
 ].forEach((controller) => {  
 // This is our instantiated class  
 // eslint-disable-next-line new-cap  
 const instance = new controller();  
 // The prefix saved to our controller  
 const path = Reflect.getMetadata('path', controller);  
 // Our `routes` array containing all our routes for this controller  
 const routes: Array<RouteDefinition> = Reflect.getMetadata('routes', controller);  
  
 // Iterate over all routes and register them to our express application  
 routes.forEach((route) => {  
 app[route.method](path+route.param, (req:express.Request, res:express.Response, next:express.NextFunction) => {  
 const result = instance[route.action]( req, res, next );  
 if (result instanceof ***Promise***) {  
 result.then((result) => result !== null && result !== undefined ? res.send(result) : next() )  
 .catch((err) => next(createError(500, err)) );  
 } else if (result !== null && result !== undefined) res.json(result);  
 });  
 });  
 });

Finally lets add the Decorations to the UserController.ts

\src\controller\UserController.ts – add the code in violet

import {getRepository, Like} from 'typeorm';  
import {NextFunction, Request, Response} from 'express';  
import {User} from '../entity/User';  
import {Controller} from '../decorator/Controller';  
import {Route} from '../decorator/Route';  
import {validate} from 'class-validator';  
  
@Controller('/users') // the base path is http://localhost:3004/users  
export default class UserController {  
 private userRepository = getRepository(User);  
  
 @Route('get') // IF the GET HTTP Request Method is used then run the action below  
 async all(request: Request, response: Response, next: NextFunction) {  
 return this.userRepository.find();  
 }  
  
 @Route('get', '/:id') // IF a param is specified then the path is http://localhost:3004/users/1  
 async one(request: Request, response: Response, next: NextFunction) {  
 return this.userRepository.findOne(request.params.id);  
 }  
  
 @Route('post') // IF the POST HTTP Request Method is used then run the action below  
 async save(request: Request, response: Response, next: NextFunction) {  
.

.

.  
 }  
  
 @Route('delete', '/:id') // IF the DELETE HTTP Request Method is used then run the action below  
 async remove(request: Request, response: Response, next: NextFunction) {  
.

.

.  
}

You can now delete \src\routes.ts as it is no longer used – remember to remove the import from \src\index.ts

Exercise: Use Postman to check the that all the GET, POST and DELETE requests still work

# Add Custom Error Handler in JSON Format

The lo4serverapi project produces JSON responses and accepts JSON requests, but when an error occurs the app crashes (example trying to delete a user a second time). We need to add custom error handlers to help with our development.

First install http-error package - <https://www.npmjs.com/package/http-error>

In the terminal run:

npm i http-error

Add handlers in index.ts to catch errors and out JSON formatted error messages

\src\index.ts – add the import at the top of the file and the code in violet just above app.listen(port)

import \* as createError from 'http-errors';

.

.

.

// catch 404 and forward to error handler  
app.use(function(req, res, next) {  
 next(createError(404));  
});  
  
// error handler  
app.use(function(err, req, res, next) {  
 res.status(err.status || 500);  
 res.json({status: err.status, message: err.message, stack: err.stack.split(/\s{4,}/)});  
});  
  
// start express server when db connection is up  
app.listen(port);  
***console***.log(`Express server has started on port ${port}. Open http://localhost:${port}/users to see results`);

Lets use the next() function in the UserController

\src\controller\UserController.ts – replace the remove action with the following

@Route('delete', '/:id') // IF the DELETE HTTP Request Method is used then run the action below  
async remove(request: Request, response: Response, next: NextFunction) {  
 const userToRemove = await this.userRepository.findOne(request.params.id);  
 response.statusCode = 204; // No Content  
 if (userToRemove) return this.userRepository.remove(userToRemove);  
 else next(); // let index.ts catch the 404 error and reply with JSON  
}

Exercise: Use Postman to make GET request to <http://localhost:3004/users/99999>? Do you get a 404 error in json format?

Exercise: Use Postman to make DELETE request to <http://localhost:3004/users/99999>? Do you get a 404 error in json format?

# Add CORS Support

Extensive explanation: <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>

Cross-Origin Resource Sharing (CORS) is an HTTP-header based mechanism that allows a server to indicate any origins (domain, scheme, or port) other than its own from which a browser should permit loading resources. CORS also relies on a mechanism by which browsers make a "preflight" request to the server hosting the cross-origin resource, in order to check that the server will permit the actual request. In that preflight, the browser sends headers that indicate the HTTP method and headers that will be used in the actual request.

We need to CORS support to the lo4serverapi app so that a web browser(like Chrome) will make a DELETE HTTP Request, otherwise the browser will refuse to make the request.

First install cors package - <https://www.npmjs.com/package/cors>

In the terminal run:

npm i cors

\src\index.ts add the import and corsOptions after the last import – add the code in violet after the bodyParser

import \* as cors from 'cors';

// cors options  
const corsOptions ={  
 origin: /localhost\:\d{4}$/i, // localhost any 4 digit port  
 credentials: true, // needed to set and return cookies  
 allowedHeaders: 'Origin,X-Requested-With,Content-Type,Accept,Authorization',  
 methods: 'GET,PUT,POST,DELETE',  
 maxAge: 43200, // 12 hours  
};

.

.

.

// setup express app here  
 app.use(bodyParser.json()); // enable body parser  
 app.use(cors(corsOptions)); // enable CORS for all handlers  
  
 // require headers 'X-Requested-With: XmlHttpRequest' and 'Accept:application/json'  
 app.use((req: express.Request, res: express.Response, next: express.NextFunction ) => {  
 if (req.xhr && req.accepts('application/json')) next();  
 else next(createError(406));  
 });  
  
 // add handler for pr-flight options request to ANY path  
 app.options('\*', cors(corsOptions));

Exercise: Try to open the app in a web browser? Do you get a 406 error?

Exercise: Add the following headers in Postman to **ALL the tabs** you have open. You need to uncheck the existing Accept header:

Graphical user interface, text, application, email

Description automatically generated

# Create Student Entity

Lets create a new enity:

In the terminal run:

typeorm entity:create -n Student

this should create the \src\entity\Student.ts

import {Column, Entity, PrimaryGeneratedColumn} from 'typeorm';  
  
@Entity()  
export class Student {

}

Lets add the properties and columns to the Student Object

import {Column, Entity, PrimaryGeneratedColumn} from 'typeorm';  
import {IsOptional, Length, MaxLength, IsNotEmpty, IsEmail, IsPhoneNumber, Matches} from 'class-validator';  
const emailOptions ={  
 allow\_display\_name: false,  
 ignore\_max\_length: false,  
 allow\_ip\_domain: false,  
 // additional validation, e.g. disallowing certain valid emails that are rejected by GMail.  
 domain\_specific\_validation: true,  
};  
  
@Entity()  
export class Student {  
 @PrimaryGeneratedColumn()  
 @IsOptional()  
 id: number;  
  
 @Column('nvarchar', {length: 50, nullable: false})  
 @Length(1, 50, {message: 'Given Name must be from $constraint1 to $constraint2 characters '})  
 @IsNotEmpty({message: 'Given Name is Required'})  
 givenName: string;  
  
 @Column('nvarchar', {length: 50, nullable: true})  
 @MaxLength( 50, {message: 'Family Name must be at most $constraint1 characters '})  
 @IsOptional()  
 familyName: string;  
  
 @Column('varchar', {length: 320, nullable: false})  
 @IsEmail(emailOptions, {message: 'Email must be in the proper format'})  
 @IsNotEmpty({message: 'Email is Required'})  
 email: string; // serves as username  
  
 @Column('varchar', {length: 15, nullable: false})  
 @Length(7, 17, {message: 'Phone Number must be from $constraint1 to $constraint2 characters'})  
 @IsPhoneNumber('CA', {message: 'Phone Number Must be a valid Canadian format'})  
 @IsNotEmpty({message: 'Phone number is Required'})  
 phone: string;  
  
 @Column('varchar', {length: 50, nullable: true})  
 @MaxLength( 150, {message: 'Address can be at most $constraint1 characters '})  
 @IsOptional()  
 address: string;  
  
 @Column('nvarchar', {length: 25, nullable: false, select: false /\* hide password from regular query\*/})  
 @Matches(/((?=.\*\d)|(?=.\*\W+))(?.\n])(?=.\*[A**-**Z])(?=.\*[a**-**z]).\*$/,  
 {message: 'Password must contain uppercase, lowercase, and numbers'})  
 @Length(8, 25, {message: 'Password must be from $constraint1 to $constraint2 characters '})  
 @IsNotEmpty({message: 'Password is Required'})  
 password: string;  
}

**Start/restart the app - typeorm will create the student table in the database:**

![A screenshot of a computer

Description automatically generatedExercise: Use the DB Navigator to add 3 rows to the student table

# Create StudentController

Lets create a new controller for the student entity

\src\constroller\StudentController.ts

import {getRepository, Like} from 'typeorm';  
import {NextFunction, Request, Response} from 'express';  
import {Student} from '../entity/Student';  
import {Controller} from '../decorator/Controller';  
import {Route} from '../decorator/Route';  
import {validate, ValidatorOptions} from 'class-validator';  
  
@Controller('/students')  
export default class StudentController {  
 private studentRepo = getRepository(Student); // Student Repository  
 // https://github.com/typestack/class-validator#passing-options  
 private validOptions: ValidatorOptions = {  
 stopAtFirstError: true,  
 skipMissingProperties: false,  
 validationError: {target: false, value: false},  
 };  
  
 @Route('get', '/:id\*?') // the \*? makes the param optional - see https://expressjs.com/en/guide/routing.html#route-paramters

async read(req: Request, res: Response, next: NextFunction) {  
 if (req.params.id) return this.studentRepo.findOne(req.params.id);  
 else {  
 const findOptions:any = {order: {}, where: []}; // prepare order and where props  
 return this.studentRepo.find(findOptions);  
 }  
 }  
  
 @Route('delete', '/:id')  
 async delete(req: Request, res: Response, next: NextFunction) {  
 const studentToRemove = await this.studentRepo.findOne(req.params.id);  
 res.statusCode = 204;  
 if (studentToRemove) return this.studentRepo.remove(studentToRemove);  
 else next();  
 }  
  
 @Route('put', '/:id')  
 async update(req: Request, res: Response, next: NextFunction) {  
 /\* PRELOAD - https://typeorm.io/#/repository-api  
 Creates a new entity from the a plain javascript object.  
 If the entity already exists in the database, then it loads it and replaces all values with the new ones from the given object,  
 and returns a new entity that is actually an entity loaded from the database with all properties replaced from the new object.  
 Note that given entity-like object must have an entity id / primary key to find entity by.  
 Returns undefined if entity with given id was not found.  
\*/  
 const studentToUpdate = await this.studentRepo.preload(req.body);  
  
 // Extra validation - ensure the id param matached the id submitted in the body  
 if (!studentToUpdate || studentToUpdate.id != req.params.id) next(); // pass the buck until 404 error is sent  
 else {  
 const violations = await validate(studentToUpdate, this.validOptions);  
 if (violations.length) {  
 res.statusCode = 422; // Unprocessable Entity  
 return violations;  
 } else {  
 return this.studentRepo.save(studentToUpdate);  
 }  
 }  
 }  
}

Now add the student controller to the list of controllers in index.ts

\src\index.ts - add the import below the last import and add the code in violet above UserController

import StudentController from './controller/StudentController';

.

.

.

// Iterate over all our controllers and register our routes  
[  
 StudentController,  
 UserController,  
].forEach((controller) => {  
 // This is our instantiated class

Exercise: Use Postman to make a GET request to localhost:3004/students, what do you get? Do you see password?

Exercise: Use Postman to make a GET request to localhost:3004/students/1, what do you get?

Exercise: Use Postman to make a PUT request localhost:3004/students/1 - Do you get an error? Why?  
    {

        "id": 2,

        "givenName": "Bob",

        "familyName": "Smithe",

        "email": "smithe@bob.ca",

        "phone": "306-555-9999",

        "address": "1 Windy Place"

    }

Exercise: Use Postman to make a PUT request localhost:3004/students/2 - Do you get a violation?   
    {

        "id": 2,

        "givenName": "Bob",

        "familyName": "Smithe",

        "email": "smithe@bob.ca",

        "phone": "306-555-9999",

        "address": "1 Windy Place"

    }

HINT: Fix the violation by adding a password to the JSON

# TypeORM Entity Manager and Repository

TypeORM uses an entity manager and repositories to create SQL statements and execute the SQL statements on database. Essentially entity manager and repository abstract the logic so developers never have to create SQL statement manually.

## What is EntityManager - <https://typeorm.io/#/working-with-entity-manager>

Using EntityManager you can manage (insert, update, delete, load, etc.) any entity. EntityManager is just like a collection of all entity repositories in a single place.

You can access the entity manager via getManager() or from Connection. Example how to use it:

**import** {getManager} **from** "typeorm";

**import** {User} **from** "./entity/User";

**const** entityManager = getManager(); // you can also get it via getConnection().manager

**const** user = **await** entityManager.findOne(User, 1);

user.name = "Umed";

**await** entityManager.save(user);

## What is Repository - <https://typeorm.io/#/working-with-repository>

Repository is just like EntityManager but its operations are limited to a concrete entity.

You can access repository via getRepository(Entity), Connection#getRepository, or EntityManager#getRepository. Example:

**import** {getRepository} **from** "typeorm";

**import** {User} **from** "./entity/User";

**const** userRepository = getRepository(User); // you can also get it via getConnection().getRepository() or getManager().getRepository()

**const** user = **await** userRepository.findOne(1);

user.name = "Umed";

**await** userRepository.save(user);

## Repository Functions - <https://typeorm.io/#/repository-api>

* metadata - The EntityMetadata of the entity managed by this repository. Learn more about transactions in Entity Metadata.  
  Example: const metadata = repository.metadata;
* createQueryBuilder - Creates a query builder use to build SQL queries.   
  Example: const users = await repository.createQueryBuilder("user")  
  .addSelect("password") .getMany();
* create - Creates a new instance of User. Optionally accepts an object literal with user properties which will be written into newly created user object  
  Example: const user = repository.create({ firstName: "Jane", lastName: "Doe"});
* preload - Creates a new entity from the given plain javascript object. If the entity already exists in the database, then it loads it (and everything related to it), replaces all values with the new ones from the given object, and returns the new entity. The new entity is an entity loaded from the database with all properties replaced from the new object.  
  Example: const user = repository.preload({ id: 1, firstName: "Timber", lastName: "Axe Is New"});
* save - Saves a given entity or array of entities. If the entity already exist in the database, it is updated. If the entity does not exist in the database, it is inserted. It saves all given entities in a single transaction (in the case of entity, manager is not transactional). Also supports partial updating since all undefined properties are skipped. Returns the saved entity/entities.  
  Example: const user = repository.save({ id: 1, firstName: "Jig", lastName: "Saw"});
* find - Finds entities that match given options.  
  Example: const users = repository.find();
* findOne - Finds first entity that matches some id or find options.  
  Example: const user = await repository.findOne(1);
* insert - Inserts a new entity, or array of entities.  
  Example: const newID = await repository.insert({ firstName: "Bob", lastName: "Smith"});

## Find Options - <https://typeorm.io/#/find-options>

All repository and manager find methods accept special options you can use to query data you need without using QueryBuilder. The find options are very robust and make it so query builder is almost never needed

## Some Basic Find Options - <https://typeorm.io/#/find-options/basic-options>

* select - indicates which properties of the main object must be selected

userRepository.find({ select: ["firstName", "lastName"] });

Will execute the query: **SELECT** "firstName", "lastName" **FROM** "user"

* where - simple conditions by which entity should be queried.

userRepository.find({ where: { firstName: "Timber", lastName: "Saw" } });

Will execute the query:

**SELECT** \* **FROM** "user"

**WHERE** "firstName" = 'Timber' AND "lastName" = 'Saw'

Using OR Operator:

userRepository.find({

where: [

{ firstName: "Timber", lastName: "Saw" },

{ firstName: "Stan", lastName: "Lee" },

],

});

Will execute the query:

**SELECT** \* **FROM** "user" **WHERE** ("firstName" = 'Timber' AND "lastName" = 'Saw')

OR ("firstName" = 'Stan' AND "lastName" = 'Lee')

* order - selection order.

userRepository.find({

order: {

name: "ASC",

id: "DESC",

},

});

Will execute the query: **SELECT** \* **FROM** "user" **ORDER** **BY** "name" **ASC**, "id" **DESC**

## Some Advanced Find options - <https://typeorm.io/#/find-options/advanced-options>

* Like

**import** { Like } **from** "typeorm";

**const** loadedPosts = **await** connection.getRepository(Post).find({

title: Like("%out #%"),

});

* Between

**import** { Between } **from** "typeorm";

**const** loadedPosts = **await** connection.getRepository(Post).find({

likes: Between(1, 10),

});

will execute following query:

**SELECT** \* **FROM** "post" **WHERE** "likes" BETWEEN 1 AND 10

* IsNull

**import** { IsNull } **from** "typeorm";

**const** loadedPosts = **await** connection.getRepository(Post).find({

title: IsNull(),

});

will execute following query:

**SELECT** \* **FROM** "post" **WHERE** "title" IS NULL

# Add Sort Functionality To Student Controller

We will use find options to allow users to sort the students by the field specified by the ***?sortby=[field]*** query string and also reverse the order if the ***&reverse=1*** query string is provided

\src\controller\StudentController.ts – ad the code in violet

const findOptions:any = {order: {}, where: []}; // prepare order and where props  
const existingFields = this.studentRepo.metadata.ownColumns.map((col)=> col.propertyName );  
const sortField:string = existingFields.includes(req.query.sortby) ? req.query.sortby : 'id';  
findOptions.order[sortField] = req.query.reverse? 'DESC' :'ASC';  
  
return this.studentRepo.find(findOptions);

The new lines of code do the following:

1. use the repository metadata function to get a list of columns in the table
2. then checks the field provided by the use in the request sortby query string against existing fields – if the sortby field is not in the existing fields then use ‘id’ as the sort field
3. if the user provided the reverse query string then set the direction to ‘DESC’ otherwise ‘ASC’

Exercise: Use Postman to GET localhost:3004/students/?sortby=address&reverse=1 - What order are the students in?

Exercise: Use Postman to GET localhost:3004/students/?sortby=email - What order are the students in?

Exercise: Use Postman to GET localhost:3004/students/?sortby=fakefield - What order are the students in?