39. Switching to TypeORM from MikroORM

#backend #typeorm #mikroorm #entity #resolver #query #mutation

Install TypeORM, uninstall MikroORM

- Due to MikroORM being too abstracted from the database and also not very user-friendly when creating many-to-one relations, we're switching to TypeORM (⚠️ **Note that** TypeORM version 0.2.25 is used in this tutorial. Many things have changed since then and these TypeORM implementations will not work for versions >= 0.3.0)

```
yarn add typeorm
yarn remove @mikro-orm/cli @mikro-orm/core @mikro-orm/migration @mikro-orm/posgresql
```

Initialize TypeORM Connection

- **Note that** TypeORM requires reflect-metadata to work, so we have to import it !

index.ts

```
import "reflect-metadata";
```

- similar to how we set up the connecttion with MikroORM, we will set up connection with TypeORM
- also create a **/src/migrations** folder, to put the custom migrations in later and point TypeOrm to look in there for migrations
- **Note that** we do not need to pass orm.em to the context anymore

index.ts

```
import "reflect-metadata";
import { COOKIE_NAME, __prod__ } from "./constants";
import { ApolloServer } from "apollo-server-express";
import connectRedis from "connect-redis";
import cors from "cors";
import express from "express";
import session from "express-session";
import Redis from "ioredis";
import { buildSchema } from "type-graphql";
import { createConnection } from "typeorm";
import { Post } from "./entities/Post";
import { User } from "./entities/User";
import { HelloResolver } from "./resolvers/hello";
import { PostResolver } from "./resolvers/post";
import { UserResolver } from "./resolvers/user";
import { MyContext } from "./types";
import path from "path";
import { Updoot } from "./entities/Updoot";

const main = async () => {
  const conn = await createConnection({
    type: "postgres",
    database: "lireddit2",
    username: "postgres",
    password: "postgres",
    logging: true,
    synchronize: true, // automatically syncs the DB so no need to run migrations - very useful in development
    migrations: [path.join(__dirname, "./migrations/*")],
    entities: [Post, User],
  });

  await conn.runMigrations();
  const app = express();

  const RedisStore = connectRedis(session);
  const redis = new Redis();

  // define CORS to avoid CORS errors
  app.use(
    cors({
      origin: "http://localhost:3000",
      credentials: true,
    })
  );

  // Initialize session storage before Apollo since it will be used from inside Apollo.
  app.use(
    session({
      name: COOKIE_NAME,
      store: new RedisStore({
        client: redis,
        disableTTL: true, // keep session alive forever
        disableTouch: true, // disable TTL reset at every touch
      }),
      cookie: {
        maxAge: 1000 * 60 * 60 * 24 * 365 * 10, // 10 years
        httpOnly: true, // prevent accessing the cookie in the JS code in the frontend
        sameSite: "lax",
        secure: __prod__, // cookie only works in https
      },
      saveUninitialized: false,
      secret: "asdfasdfasdf", // used to sign cookie - should actually be hidden in an env variable
      resave: false,
    })
  );

  const apolloServer = new ApolloServer({
    schema: await buildSchema({
      resolvers: [HelloResolver, PostResolver, UserResolver],
      validate: false,
    }),
    context: ({ req, res }: MyContext) => ({ req, res, redis }), // context is shared with all resolvers
  });

  apolloServer.applyMiddleware({
    app,
    cors: false,
  });

  app.listen(4000, () => {
    console.log("server started on localhost:4000");
  });

main().catch((err) => {
  console.log(err);
});
```

Update Entities from MikroORM to TypeORM

- typeorm.io/#entities
- User and Post entities were tagged with the MikroOrm's @ attributes. We update them to TypeORM as follows
- **Note that** there are specific attributes @CreateDateColumn() and @UpdateDateColum() for date management
- BaseEntity allows Post.find(), Post.insert(), some easy command to be used in SQL
- With TypeOrm we don't need to specify `{ type: "text" }` for string types

/entities/Post.ts

MikroORM	TypeORM

import { Field, ObjectType } from "type-graphql";
import { Entity, PrimaryKey, Property } from "@mikro-orm/core";
 	
@ObjectType() // graphQL
@Entity() // mikro-orm
export class Post {
  @Field()
  @PrimaryKey()
  id!: number;
 	
  @Field(() => String) // explicitly set type for GraphQL
  @Property({ type: 'date' }) // explicitly set type for MikroORM
  createdAt = new Date();
 	
  @Field(() => String)
  @Property({ type: 'date', onUpdate: () => new Date() })
  updatedAt = new Date();
	import { Field, ObjectType } from "type-graphql";
	import { BaseEntity, Column, CreateDateColumn, Entity, PrimaryGeneratedColumn, UpdateDateColumn } from "typeorm";
	
	@ObjectType() // graphQL
	@Entity() // typeorm
	export class Post extends BaseEntity {
	  @Field()
	  @PrimaryGeneratedColumn()
	  id!: number;
	
	  @Field(() => String) // explicitly set type for GraphQL
	  @CreateDateColumn()
	  createdAt: Date;
	
	  @Field(() => String)
	  @UpdateDateColumn()
	  updatedAt: Date;
	
	  @Field()
	  @Column()

```
  @Field()
  @Property({ type: 'text'})
  title!: string;
}
```

```
  title!: string;
}
```

/entities/User.ts

MikroORM

```
import { Field, ObjectType } from "type-graphql";
import { Entity, PrimaryKey, Property } from "@mikro-orm/core";


@ObjectType()
@Entity()
export class User {
  @Field()
  @PrimaryKey()
  id!: number;

  @Field(() => String)
  @Property({ type: "date" })
  createdAt = new Date();

  @Field(() => String)
  @Property({ type: "date", onUpdate: () => new Date() })
  updatedAt = new Date();

  @Field()
  @Property({ type: "text", unique: true })
  username!: string;

  @Field()
  @Property({ type: "text",
```

TypeORM

```
import { Field, ObjectType } from "type-graphql";
import { BaseEntity, Column, CreateDateColumn, Entity, PrimaryGeneratedColumn, UpdateDateColumn } from "typeorm";

@ObjectType()
@Entity()
export class User extends BaseEntity {
  @Field()
  @PrimaryGeneratedColumn()
  id!: number;

  @Field(() => String)
  @CreateDateColumn()
  createdAt: Date;

  @Field(() => String)
  @UpdateDateColumn()
  updatedAt: Date;

  @Field()
  @Column({ unique: true })
  username!: string;

  @Field()
  @Column({ unique: true })
  email!: string;
unique: true })
  email!: string;

  @Property({ type: "text" })
  password!: string;
}
```

```
  @Column()
  password!: string;
}
```

Update Context

- since we do not need to pass orm.em to the context anymore, we delete it from MyContext

types.ts

```
export type MyContext = {
  // Not needed anymore, we delete this --->  em:
EntityManager<IDatabaseDriver<Connection>>;
  req: ExtendedRequest;
  res: Response;
  redis: Redis; // to be added during (11)
};
```

Update Post Resolver

- Since we do not use em.orm anymore, we update the Resolvers accordingly

/resolvers/post.ts

```
import { Post } from "../entities/Post";
import { MyContext } from "src/types";
import { Arg, Ctx, Int, Mutation, Query, Resolver } from "type-graphql";

@Resolver()
export class PostResolver {
  @Query(() => [Post]) // [Post] is how we define arrays in return type for the
resolver
  async posts(): Promise<Post[]> {
    return Post.find()
  }

  @Query(() => Post, { nullable: true })
  post(@Arg("id") id: number): Promise<Post | undefined> {
    return Post.findOne(id)y;
  }

  @Mutation(() => Post)
  async createPost(@Arg("title") title: string): Promise<Post> {
    return Post.create({title}).save();
  }

  @Mutation(() => Post, { nullable: true })
  async updatePost(
    @Arg("id") id: number, // here we ommitted type declaration in @Arg - type inference works for Int and String
    @Arg("title", () => String, { nullable: true }) title: string // here we explicitly set type since we want to make it nullable
  ): Promise<Post | null> {
    const post = await Post.findOne(id);

    if (!post) {
      return null;
    }

    if (typeof title !== "undefined") {
      post.title = title;
      await Post.update({id}, {title});
    }

    return post; // this is actually wrong and returns the unmodified post. we'll fix it later
  }

  @Mutation(() => Boolean)
  async deletePost(@Arg("id") id: number): Promise<boolean> {
    const post = await em.findOne(Post, { id });

    if (!post) {
      return false;
    }

    await Post.delete(id);
    return true;

  }
}
```

Update User Resolver

- Note that we can use User.findOne(id) since id is the primary key
- When searching with a key that is ***not*** the primary key we use { where : key : value }

  e.g. `User.findOne({ where: email })` or `User.findOne({ where: { email : userNameOrEmail })`

/resolvers/user.ts

```
import { User } from "../entities/User";
import { MyContext } from "src/types";
import {Arg, Ctx, Field, Mutation, Query, Resolver} from "type-graphql";
import argon2 from "argon2";
import { UsernamePasswordInput } from "./UsernamePasswordInput";
import { validateRegister } from "../utils/validateRegister";
import v4 from "uuid"
import { getConnection } from "typeorm";

@ObjectType() // ObjectTypes are returned from Queries and Mutations
class FieldError {
  @Field()
  field: string; // which field the error is about
  @Field()
  message: string; // error message
}

@ObjectType()
class UserResponse {
  @Field(() => [FieldError], { nullable: true })
  errors?: FieldError[];

  @Field(() => User, { nullable: true })
  user?: User;
}

@Resolver()
```

```
export class UserResolver {
  @Mutation(() => UserResponse)
  async changePassword(
    @Arg("token") token: string,
    @Arg("newPassword") newPassword: string,
    @Ctx() { redis, req }: MyContext
  ): Promise<UserResponse> {
    if (newPassword.length <= 2) {
      return {
        errors: [
          {
            field: "newPassword", // must match the name of the field on front-end
            message: "Length must be greater than 3",
          },
        ],
      };
    }

    const tokenKey = FORGOT_PASSWORD_PREFIX + token;
    const userId = await redis.get(tokenKey); // retrieve value for token from redis
    if (!userId) {
      return {
        errors: [
          {
            field: "token",
            message: "Token expired",
          },
        ],
      };
    }

    const userIdNum = parseInt(userId);
    const user = await User.findOne(parseInt(userIdNum));

    if (!user) {
      return {
        errors: [
          {
            field: "token",
            message: "User no longer exists",
          },
        ],
      };
    }

    await User.update(
      { id: userIdNum },
      { password: await argon2.hash(newPassword) }
    ); // change pw in db

    await redis.del(tokenKey); // delete token so it can't be reused
    req.session.userId = user.id; // log the user in
    return { user };
  }

  @Mutation(() => Boolean)
  async forgotPassword(
    @Arg("email") email: string,
    @Ctx() { redis }: MyContext
  ): Promise<Boolean> {
    const user = await User.findOne({ where: email }); // email not primary key, so we have to use "where"
    if (!user) {
      // the email is not in the db
      return true; //  don't let the person know that the email is not in the db
    }

    const token = v4(); // token for resetting pw

    // save token to redis with value userId, expires in 1 day
    await redis.set(
      FORGOT_PASSWORD_PREFIX + token, // redis key
      user.id, // value
      "ex", // expiry mode
      1000 * 60 * 60 * 24 // expiration duration - 24 hours
    );
    const resetLink = `<a href="http://localhost:3000/change-password/${token}">Reset password</a>`;
    sendEmail(email, "Reset Password", resetLink);
    return true;
  }

  @Query(() => User, { nullable: true })
  me(@Ctx() { req }: MyContext) {
    // you are not logged in
    if (!req.session.userId) {
      return null;
    }

    return User.findOne(req.session.userId);
  }

  @Mutation(() => Boolean)
  async logout(@Ctx() { req, res }: MyContext): Promise<Boolean> {
    // clear the user's cookie
    res.clearCookie(COOKIE_NAME);

    // clear the redis record
    return new Promise(
      (
        resolve // remove the session from redis
      ) =>
        req.session.destroy((err) => {
          if (err) {
            console.log(err);
            resolve(false);
            return;
          }
          resolve(true);
        })
    );
  }

  @Mutation(() => UserResponse)
  async login(
    @Arg("usernameOrEmail") usernameOrEmail: string,
    @Arg("password") password: string,
      @Ctx() { req }: MyContext
    ): Promise<UserResponse> {
      const user = await User.findOne(
        usernameOrEmail.includes("@")
          ? { where: { email: usernameOrEmail } }
          : { where: { username: usernameOrEmail } }
      );
      if (!user) {
        return {
          errors: [
            {
              field: "usernameOrEmail",
              message: "That username or email does not exist",
            },
          ],
        };
      }

      const isPasswordValid = await argon2.verify(user.password, password);
      if (!isPasswordValid) {
        return {
          errors: [
            {
              field: "password",
              message: "Incorrect password",
            },
          ],
        };
      }

      req.session.userId = user.id; // created new type for req in types.ts to make this work, so the session can store the userId

      return { user };
    }

    async register(
      @Arg("options") options: UsernamePasswordInput, //  let typescript infer type UsernamePasswordInput
      @Ctx() { req }: MyContext
```

```
): Promise<UserResponse> {
  const errors = validateRegister(options);
  if (errors) {
    return { errors };
  }

  const hashedPassword = await argon2.hash(options.password);

  let user;
  try {
    /* Same opeartion Using .create - but may return undefined */
    // user = await User.create({
    //   username: options.username,
    //   password: hashedPassword,
    //   email: options.email,
    // }).save();

    const result = await getConnection()
      .createQueryBuilder()
      .insert()
      .into(User)
      .values({
        username: options.username,
        password: hashedPassword,
        email: options.email,
      })
      .returning("*")
      .execute();
    user = result.raw[0];
  } catch (err) {
    // duplicate username error
    if (err.code === "23505") {
      return {
        errors: [
          {
            field: "username",
            message: "That username is already taken",
          },
        ],
      };
      }
    }

    req.session.userId = user.id; // logs in the user (by sending cookie to
browser)
    return { user };
  }
}
```