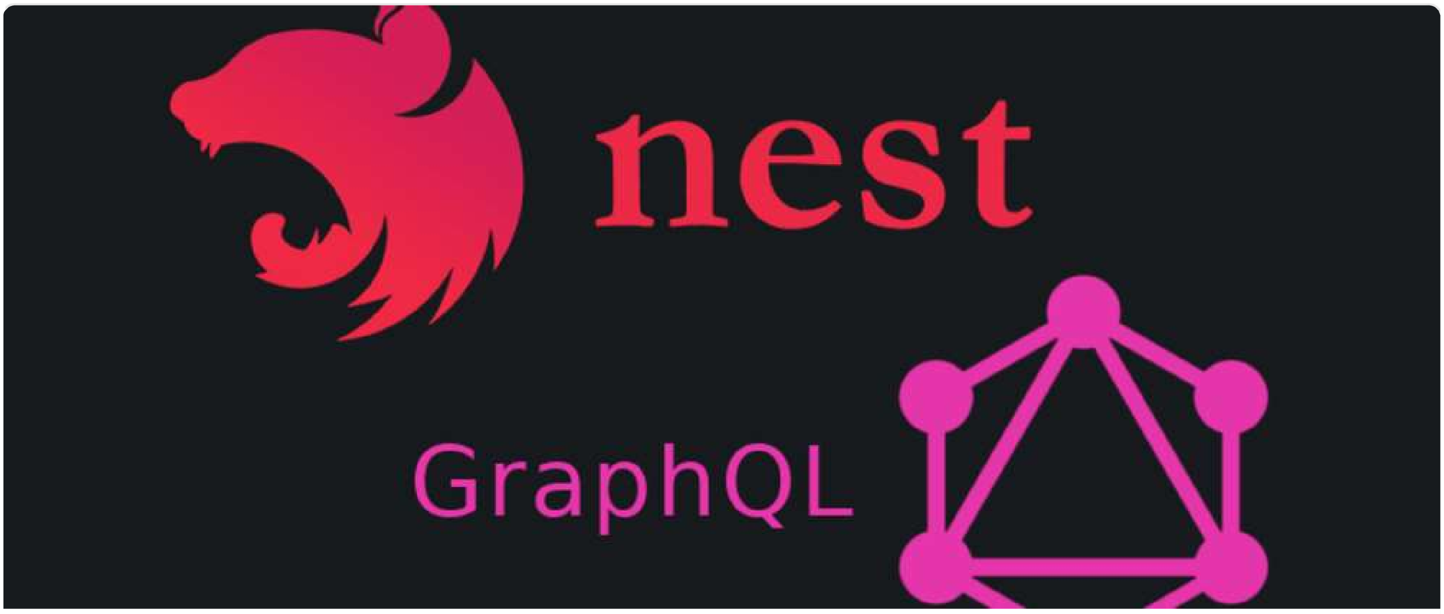


3logy
Posted on 3 de mai. de 2020

Signup and Login CQRS Pattern with Nest JS, Passport and GraphQL

#win10 #nestjs #graphql #passport

Table Of Contents

Short introduction

Nest JS is a framework for building efficient, scalable Node.js server-side applications. It uses progressive JavaScript and fully supports TypeScript. My decision to adopt this framework was largely influenced by the very good documentation 🍍 and especially the similarity of implementation of the API with the Spring framework 💎.

I will try to be quite concise and precise in this tutorial. It will involve implementing a registration and login process. I would not go into details in the **installation of tools**

or frameworks. At the End of this tutorial i put a link to the github repository.

Have Fun! ⏳

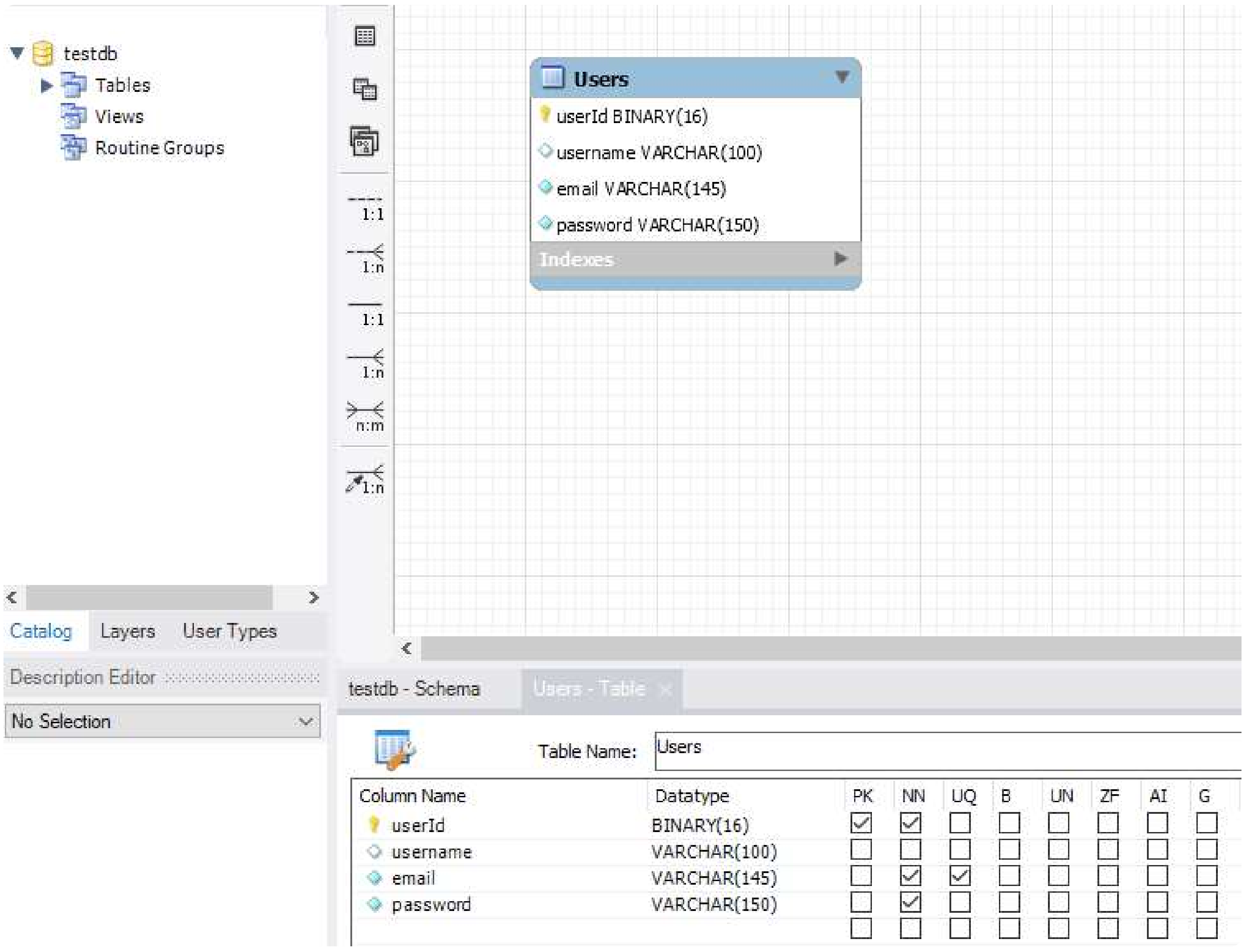
Tools

I used the following tools in this tutorial:

- Windows 10 pro
- VS Code
- MySQL Workbench + MySQL
- git 2.20
- CMDER
- Node JS 13.13
- NPM 6.14.4
- Nest 7.1.2

Setup MySQL database

After installing mysql and possibly mysql workbench, we will create a database with a **Users** table in it.



Users
userId BINARY(16)
username VARCHAR(100)
email VARCHAR(145)
password VARCHAR(150)

Table Name: Users

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G
userId	BINARY(16)	✓	✓						
username	VARCHAR(100)								
email	VARCHAR(145)		✓	✓					
password	VARCHAR(150)		✓						

Scaffolding project with cli

Open the cmder, create a new project and choose npm at the prompt :

```
λ nest new cqrs-auth-db
λ npm i --save @nestjs/core @nestjs/common rxjs reflect-metadata
```

This command will automatically create default folders and files. Now we will add an .env file in the root of the project. 🔔

```
λ touch .env
```

We will come back to this later.
In nestjs each application has at least one module, a root module. We will add a new "users" module to our project.

```
λ nest g mo users
CREATE src/users/users.module.ts (82 bytes)
UPDATE src/app.module.ts (312 bytes)
```

Use schema first GraphQL

We need to install the apollo server to use graphql.

```
λ npm install apollo-server-express
```

For implementing the GraphQL, i choose the **schema first** way. So we will first generate the following files:

- src/users/users.resolver.ts
- src/users/users.graphql
- src/app.graphql

In the **app.graphql** we will define input, type, mutation and schema.

```
input Signup {
    username: String!,
    email: EmailAddress!,
    password: String!,
    passwordConfirm: String!,
}

type SignupResponse {
  username: String!,
  email: String!,
}

type AuthPayload {
    Authorization: String!
    user: String!
    expiresIn: String!
}

type Mutation {
  signup(input: Signup!): SignupResponse!
  login(username: EmailAddress!, password: String!): AuthPayload!
}

schema {
  query: Query
}
```

Settings the application

In this application we will need some libraries and settings for passport, cqrs, graphql, configuration and typeorm.

```
λ npm i --save @nestjs/graphql graphql-tools graphql
λ npm i --save @nestjs/passport passport passport-local
λ npm i --save-dev @types/passport-local
λ npm i --save @nestjs/typeorm typeorm mysql
λ npm i --save @nestjs/cqrs
λ npm i --save @nestjs/config
```

Now, we will generate from the graphql schemas the classes in the necessary javascript file **graphql.schema.ts**. Add this configuration to the **import:[]** in the **app.module.js**:

```
GraphQLModule.forRootAsync({
      useFactory: () => ({
        context: ({ req }) =>  ({ req }),
        typeDefs: [],
        resolvers: [],
        typePaths: ['./**/*.graphql'],
        installSubscriptionHandlers: true,
        definitions: {
          path: join(process.cwd(), 'src/graphql.schema.ts'),
          outputAs: 'class',
        },
      }),
    }),
```

For typeorm, we will need to configure access to the mysql database. We will create a **TypeOrmConfigService** class which will contain this configuration for the TypeOrmModule:

```
@Injectable()
export class TypeOrmConfigService implements TypeOrmOptionsFactory {

  constructor(private configService: ConfigService ) {}

  createTypeOrmOptions(): TypeOrmModuleOptions {
    return {
      type: 'mysql',
      host: process.env.TYPEORM_HOST,
      port: Number(process.env.TYPEORM_PORT),
      username: process.env.TYPEORM_USERNAME,
      password: process.env.TYPEORM_PASSWORD,
      database: process.env.TYPEORM_DATABASE,
      entities: [__dirname + '/../**/*.entity{.ts,.js}'],
      synchronize: false,
      migrationsRun: true,
      migrationsTableName: "custom_migration_table",
      migrations: [__dirname + "migration/*.js"],
      "cli": {
        "migrationsDir": "migration"
      },
      autoLoadEntities: true,
      logging: ["query", "error"],
    };
  }
}
```

Then, add this configuration to the **import:[]** in the **app.module.js**:

```
TypeOrmModule.forRootAsync({
      imports: [ConfigModule],
      useClass: TypeOrmConfigService,
      inject: [ConfigService],
 }),
```

Now it's time to go back to the **.env** file, did you remember? 😏
But before we need to add the ConfigModule to the root module **app.module.ts**.

```
imports:[.., ConfigModule.forRoot({ isGlobal: true, }), ..]
```

We can add this configuration to **.env**, replace dbname and dbpass with your actual configuration:

```
TYPEORM_HOST = "localhost"
TYPEORM_USERNAME = "dbname"
TYPEORM_PASSWORD = "dbpass"
TYPEORM_DATABASE = "testdb"
TYPEORM_PORT = 3306
```

If you run **npm run start:dev** you will be able to see the graphql interface at **localhost:3000/graphql**

The routing with GraphQL is a little bit different. We need a class with a Resolver decoration:
app.resolver.ts

```
@Resolver ('App')
export class AppResolver {

    constructor(
        private commandBus: CommandBus,
    ) {}

    @Mutation('signup')
    public async signup(@Body('input') input: SignUp ) {

        try {
            return await this.commandBus.execute(
                new CreateUserCommand(input.username,
                   input.email, input.password));
        } catch (errors) {
            console.log("Caught promise rejection (validation failed). Errors: ", errc
        }
    }
}
```

From here now we can send a request like:

```
mutation {
  signup(
    input: {
      username:"demo",
      email:"email@mail.com",
      password:"!?94de807286baDbpass"}
  ) {
    email, username
  }
}
```

Command, Event and Sagas

There are already a multitude of tutorials on DDD and CQRS. I will limit myself here to the implementation of the cqrs pattern with nestjs and graphql.

User Register

In the Resolver class we have the commandBus which execute a Command. If you want to know more about read the documentation.

The following folders are created:

- src/users/commands/handlers
- src/users/commands/impl
- src/users/events/handlers
- src/users/events/impl
- src/users/repositories
- src/users/sagas
- src/users/user.entity.ts

Then we create the command Interface and his implementation:

create-user.command.ts

```
export class CreateUserCommand {
    constructor(
      public readonly username: string,
      public readonly email: string,
      public readonly password: string,
    ) {}
}
```

create-user.handler.ts

```
@CommandHandler(CreateUserCommand)
export class CreateUserHandler implements ICommandHandler<CreateUserCommand> {
    constructor(
        private readonly eventBus: EventBus,
    ) { }

    async execute(command: CreateUserCommand) {

        const { username, email, password } = command;
        const userRepository = getCustomRepository(UserRepository);

        const user = userRepository.create();
        user.userId = await this.convertStringToBinary(uuidv4());
        user.username = username;
        user.email = email;
        user.password = password;

        const userDB: Users = await userRepository.save(user);


        this.sendEvent(userDB.userId, this.eventBus);

        return userDB;
    }

    private async sendEvent(userId: Buffer, eventBus: EventBus) {
        if(userId !== undefined) {
            console.log("send event UserCreatedEvent");
            eventBus.publish(
                new UserCreatedEvent(
                    Buffer.from(userId).toString('hex')));
        }
    }

    private async convertStringToBinary(uuid: string): Promise<Buffer> {
        const guid = Buffer.alloc(16);
        guid.write(uuid);
        return guid;
    }
}
```

If you now send a **valid** signup you will end up with response json:

```
{
  "data": {
    "signup": {
      "email": "email@mail.com",
      "username": "demo"
    }
  }
}
```

User Login

For the login we have to use Guards. NestJS already provide a really good documentation about it.
In this example i will show you how Passport Guard and GraphQL can work together.

First we have to extends the passport **AuthGuard** and then manually built the *request object*.

```
@Injectable()
export class LocalAuthGuard extends AuthGuard('local') {

    constructor() {
        super();
    }

    async canActivate(context: ExecutionContext): Promise<boolean> {
        await super.canActivate(context);
        const ctx = GqlExecutionContext.create(context);
        const request  = ctx.getContext();
        request.body = ctx.getArgs();
        return true;
    }

    getRequest(context: ExecutionContext) {
        const ctx = GqlExecutionContext.create(context);
        const request  = ctx.getContext();
        request.body = ctx.getArgs();
        return request;
    }
}
```

In the root resolver we have to add the login function and use our custom decorator on it. If you want to use JWT to generate a token please read the documentation

app.resolver.ts

```
...
@UseGuards(LocalAuthGuard)
    @Mutation('login')
    public async login(@CurrentUser() req ): Promise<AuthPayload> {

        return {
            email: req.email,
        };
    }
...
```

To validate the username and password combo, the passport uses a pattern called strategy. Here is how our stragety looks like:

local.strategy.ts

```
@Injectable()
export class LocalStrategy extends PassportStrategy(Strategy) {
  constructor(
    private commandBus: CommandBus,
  ) { super(); }

  async validate(email: string, password: string): Promise<any> {
    const user = await this.commandBus.execute(
      new PerformLoginCommand(email, password)
    );
    if (!user) {
      throw new UnauthorizedException();
    }
    return user;
  }
}
```

Here we also use as you see the cqrs using a command **PerformLoginCommand**. We need 2 classes a handler and the interface.
I keep it simple, but this class could be more complicated if you deal with hashed password for example.
perform-login.handler.ts

```
@CommandHandler(PerformLoginCommand)
export class PerformLoginHandler implements ICommandHandler<PerformLoginCommand> {

    async execute(command: PerformLoginCommand) : Promise<any> {
        const { email, password } = command;
        const user = getRepository(Users)
            .find({email: email, password: password});
        return user;
    }
}
```

Now you can easily sent the request:

```
mutation {
  login(
    username:"email@mail.com",
    password:"!?94de807286baDbpass",
  ) {
    email,
  }
}
```

and get the email as response.

NestJ brings very good tools to work with the CQRS pattern. In the github repo, you can also see how I implemented the Saga and Event model.

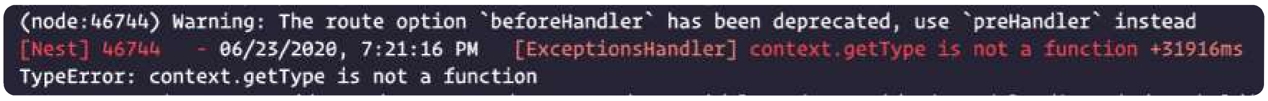
[Final Repo](#) 🎉
[bit.ly link](#)

Discussion (8)

Jorge Guerra • Jun 24 '20

Hi 3logy, thanks for this amazing article.

I'm trying to implement it but, I'm getting this error: **TypeError: context.getType is not a function**



```
(node:46744) Warning: The route option `beforeHandler` has been deprecated, use `preHandler` instead
[Nest] 46744   - 06/23/2020, 7:21:16 PM   [ExceptionsHandler] context.getType is not a function +31916ms
TypeError: context.getType is not a function
```

What could be happening? Or what might have I forgotten?

3logy • Jun 30 '20

Hi Jorge, sorry for the late, I was in vacation. Did you fix the issue?

Best regards

Jorge Guerra • Jul 10 '20

Brooo I've alrady solved it. jajaja it was actually tough.

Takis Koumoutsakos • Feb 17

Please share how you solved it

Jorge Guerra • Feb 19

Here it is:

```
export const Context = createParamDecorator(
    (data, [root, args, ctx, info]) => ctx
);
```

In @nestjs/core 6.0.0 you can access the context like that.

Does it solve your problem?

Takis Koumoutsakos • Feb 19

Thanks for your reply Jorge, will try later.

Jorge Guerra • Jul 1 '20

No problem 3logy. And no, haven't solved It yet, decided to take a rest after try for two days hahaha. Maybe I'll try it tomorrow. I'll write another comment here letting you know how it went.

Николай • Jul 3

Hi 3logy, could you suggest how would your LocalStrategy looks in case when service has to explain one of two possible reasons why login has failed

- incorrect login/pass
- expired subscription ? Thanks in advance

Code of Conduct • Report abuse

3logy

Convergent and divergent Thinker. I like to go deep into code, details but at the same time to have a holistic understanding of things (#architecture, #macroeconomics)

LOCATION
Germany

EDUCATION
MBA candidate & Bachelor of Sciences

WORK
Technical Lead Developer

JOINED
23 de abr. de 2019

Trending on DEV Community 🔥

What was your win this week?

#discuss #weeklyretro

Is "AI" generated music finally useful? [for Indie Game Devs]

#webdev #machinelearning #discuss #gamedev

Is Microservices a good choice ?

#help #discuss #microservices #architecture