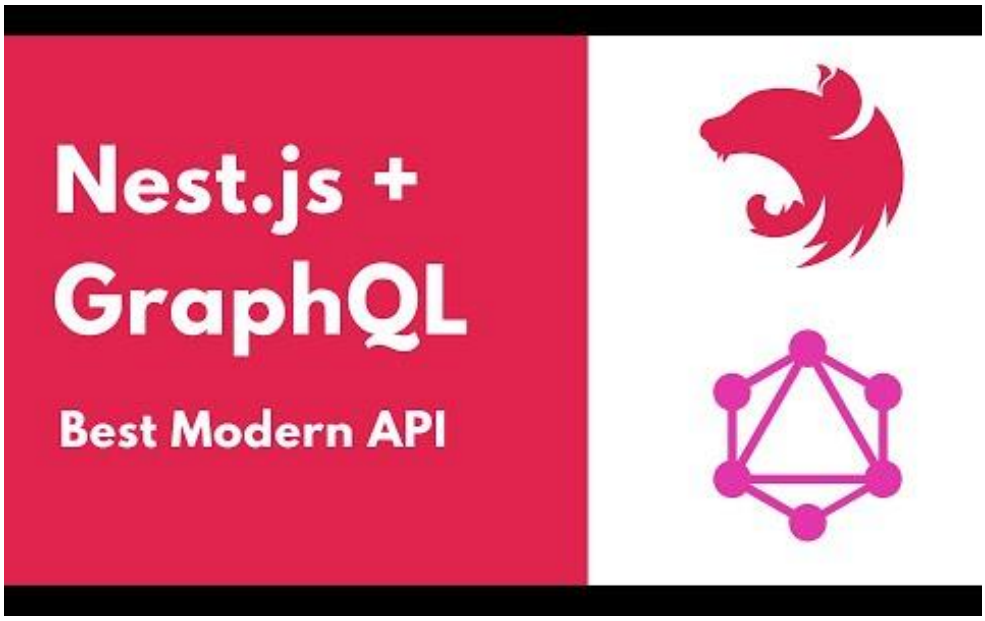
Bytes Technolab

BytesTechnolab Weekly Digest

Setting up a GraphQL project using Nest.js, TypeORM (Code-first approach)



by Dharmaraj Jadeja on February 07

Today we are going to learn about Nest.js, what it actually is, why it is so popular and create a simple GraphQL project using Nest.js.

In this article we will be focusing on basic understanding of Nest.js and how it is different from other Node framework, for that we will be creating a very simple project where Users can **create** their account. After this tutorial you would have a better understanding of:

- Basic understanding of Nest.js
- How to implement GraphQL with Nest.js
- Basic understanding of typeorm in Nest.js

Prerequisites

- Basic understanding of Node and Javascript
- Familiarity with GraphQL resolvers and schema
- Postgres Database and its queries.
- Database tool installed like DBevear, PG Admin,etc

What is Nest.js?

Nest (NestJS) is a modern Node.js framework. It has built-in Typescript support so it supports all of the Typescript features. The reason that makes Nest.js different from other Node.js frameworks is that it combines factors of OOP (object oriented Programming), FP (functional Programming), and FRP (practical Reactive Programming) all in one place, which makes developing complex apps much easier.

Nest.js has official support for the most popular HTTP REST and the latest GraphQL api's which makes the framework more dynamic. For REST api's it has built-in support of Express (The most popular) and Fastify too. It also has built in support for WebSockets and Microservices too. As we can see we can create both Monolithic and Microservices app using a single framework, which is quite extraordinary.

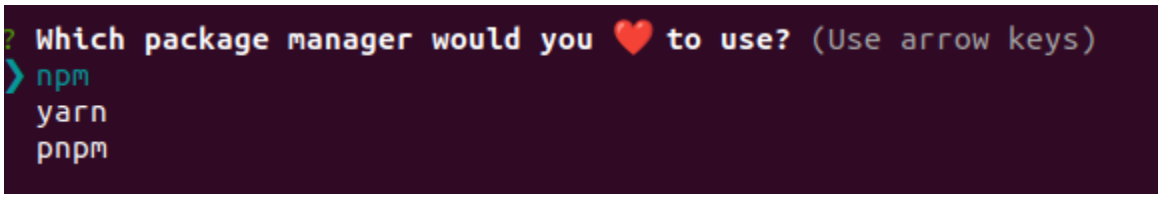
Nest.js Installation

Let's make use of the powerful nest cli, run the following command which will automatically create a new nest js project directory with required files.

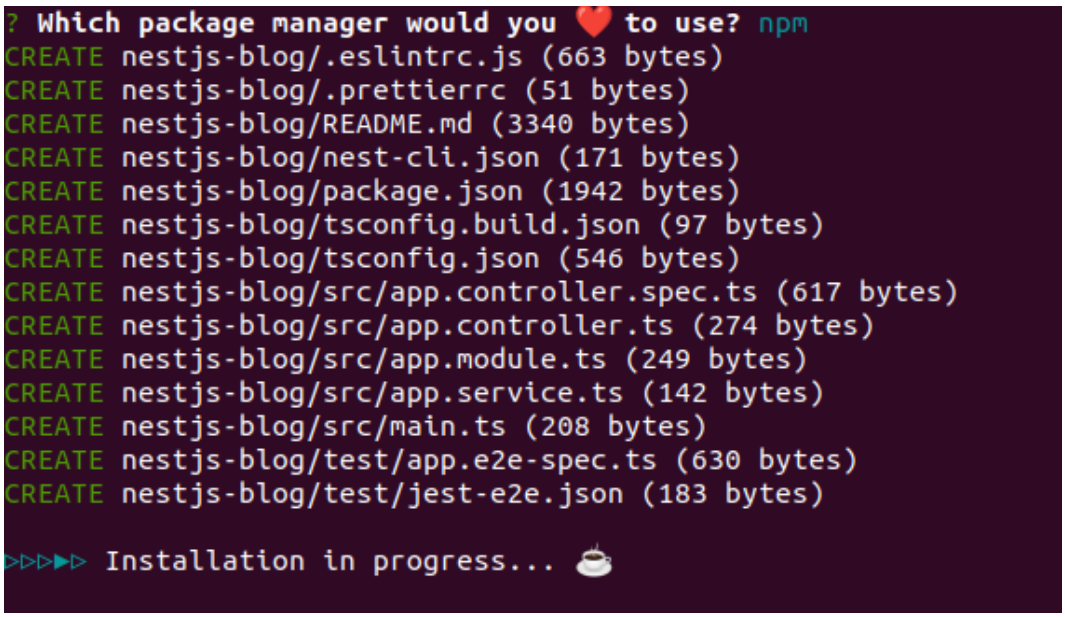
```
$ npm i -g @nestjs/cli
$ nest new nest-graphql
```

After entering the above code you will be prompted few options:

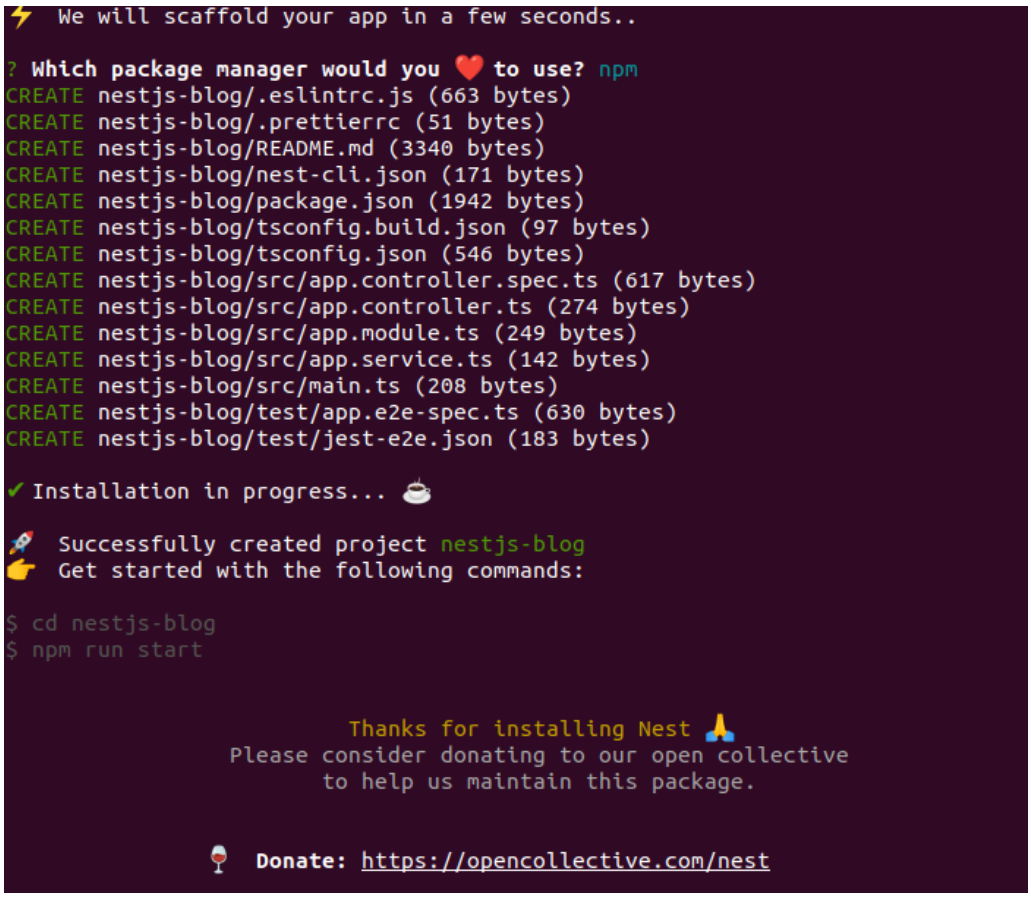
1. Which package manager would you use? We will be using the most famous package **npm**



2. After clicking next you wait for the installation to complete.



3. After installation is completed you should see the following screen:



Once the installation is complete, you will see that a new folder with the name "nest-graphql" has been created. This would be the main project folder. Now let's move on to the next section.

Harnessing the power of TypeScript & GraphQL

GraphQL is a powerful query language. It's an elegant approach that solves many problems typically found with REST APIs. GraphQL combined with TypeScript makes the development easy.

Before moving forward, it is necessary that you have a basic understanding of GraphQL and typescript.

Now let's focus on how to work with the built-in @nestjs/graphql module. It can be configured to use Apollo server (with the @nestjs/apollo driver) and Mercurius (with the @nestjs/mercurius). Nest.js provides official simple and easy integrations for these GraphQL packages.

Server Installation

We'll be implementing Express and Apollo server, so let's start by installing the required packages:

```
$ cd nest-graphql
$ npm i @nestjs/graphql @nestjs/apollo graphql apollo-server-express
apollo-server-core
```

Now let's understand what we installed.

- **@nestjs/graphql** is the graphql package which allows us to use graphql in our nest project.
- **@nestjs/apollo** is the nest's apollo package which will be used to host the graphql server.
- **graphql** is used to install graphql in our system.
- **apollo-server-express** community-maintained open-source GraphQL server that works with many Node.js HTTP server frameworks.
- **apollo-server-core** to use Apollo Sandbox as a GraphQL IDE for local development.

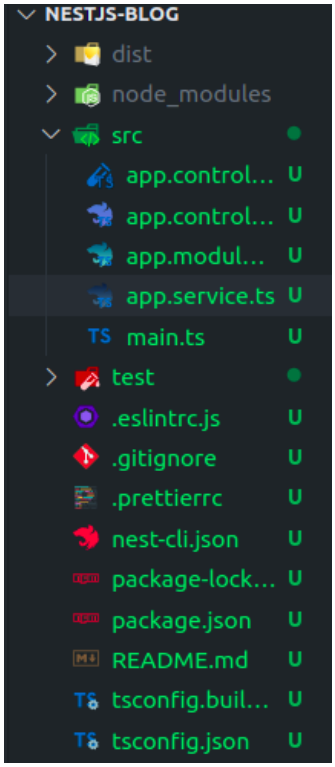
Nest gives 2 approaches to constructing GraphQL applications, the **code first** and the **schema first** methods.

The main difference between **code first** and **schema first** is that in **code first** approach we don't have to worry about managing the graphql schema, nest automatically handles the schema generation. For more details on these approaches refer to this official documentation.

For this article we'll be using the **code first** approach.

Let's Get Started with GraphQL and Typescript

Now we are all set up with the basic boilerplate for our nest js project. You should have the following folder structure automatically generated by the nest cli:



Now open the **app.module.ts** file, here we will configure our GraphQL server configuration.

To use the code first approach, start by adding the **autoSchemaFile** property to the options object.

```
import { ApolloDriver, ApolloDriverConfig } from '@nestjs/apollo';
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';
import { ApolloServerPluginLandingPageLocalDefault } from 'apollo-server-core';
import { join } from 'path';
import { AppController } from './app.controller';
import { AppService } from './app.service';
```

```
@Module({
 imports: [
   GraphQLModule.forRoot<ApolloDriverConfig>({
     driver: ApolloDriver,
     autoSchemaFile: join(process.cwd(), '/schema.gql'),
     sortSchema: true,
     playground: false,
     plugins: [ApolloServerPluginLandingPageLocalDefault()],
   }),
 ],
 controllers: [AppController],
 providers: [AppService],
})
export class AppModule {}
```

The **autoSchemaFile** property value is the path where your automatically generated schema will be created. Alternatively, the schema can be generated on-the-fly in memory. To enable this, set the autoSchemaFile property to **true.**

By default, the types in the generated schema will be in the order they are defined in the included modules. To sort the schema lexicographically, set the **sortSchema** property to **true.**

Modules & Resolvers

Resolvers provide the instructions for turning a GraphQL operation (a query, mutation, or subscription) into data. They return the same shape of data we specify in our schema -- either synchronously or as a promise that resolves to a result of that shape. Typically, you create a resolver map manually.

The **@nestjs/graphql** package, on the other hand, generates a resolver map automatically using the metadata provided by decorators you use to annotate classes.

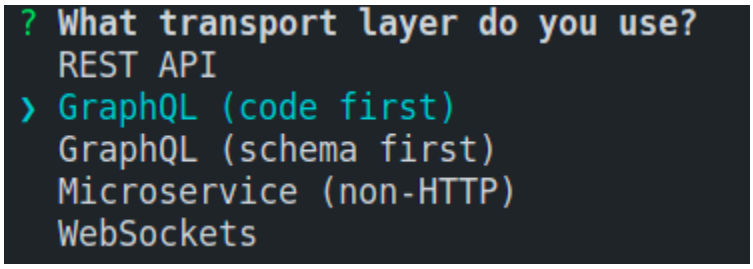
So let's start creating our first resolver. We will be making use of the powerful cli provided by nest. Go to the src folder and run the following command:

```
$ cd src/
$ nest g resource user --no-spec
```

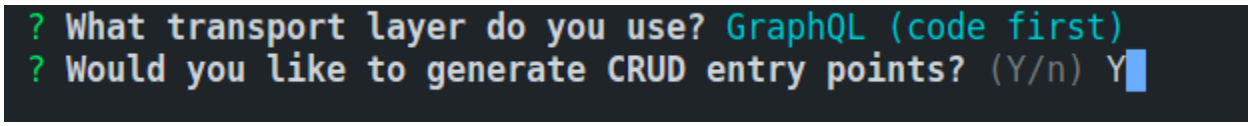
Here we are passing **--no-spec** option to exclude the **.spec** files which are used for testing purposes and are not at all required to run the application.

Follow the instructions as below:

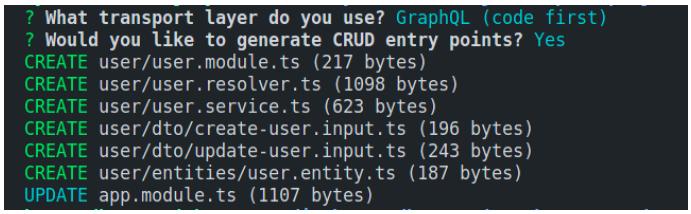
1. Select GraphQL (code first) option:



2. Select



3. Success!



The above command has automatically generated all the required file and added the files in the **app.module.ts.**

Now add the following code inside the User Resolver class:

```
@Query((returns) => String)
  helloWorld() {
    return 'Hello World';
  }
```

The file **user.resolver.ts** should look like this:

```
import { Resolver, Query, Mutation, Args, Int } from '@nestjs/graphql';
import { UserService } from './user.service';
import { User } from './entities/user.entity';
import { CreateUserInput } from './dto/create-user.input';
```

```
import { UpdateUserInput } from './dto/update-user.input';

@Resolver(() => User)
export class UserResolver {
  constructor(private readonly userService: UserService) {}

  @Mutation(() => User)
  createUser(@Args('createUserInput') createUserInput: CreateUserInput) {
    return this.userService.create(createUserInput);
  }

  @Query(() => [User], { name: 'user' })
  findAll() {
    return this.userService.findAll();
  }

  @Query(() => User, { name: 'user' })
  findOne(@Args('id', { type: () => Int }) id: number) {
    return this.userService.findOne(id);
  }

  @Mutation(() => User)
  updateUser(@Args('updateUserInput') updateUserInput: UpdateUserInput) {
    return this.userService.update(updateUserInput.id, updateUserInput);
  }

  @Mutation(() => User)
  removeUser(@Args('id', { type: () => Int }) id: number) {
    return this.userService.remove(id);
  }

  @Query((returns) => String)
  helloWorld() {
    return 'Hello World';
  }
}
```

app.module.ts:

```
import { ApolloDriver, ApolloDriverConfig } from '@nestjs/apollo';
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';
import { ApolloServerPluginLandingPageLocalDefault } from
```

```
'apollo-server-core';
import { join } from 'path';

import { AppController } from './app.controller';
import { AppService } from './app.service';
import { UserModule } from './user/user.module';

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloDriverConfig>({
      driver: ApolloDriver,
      autoSchemaFile: join(process.cwd(), '/schema.gql'),
      sortSchema: true,
      playground: false,
      plugins: [ApolloServerPluginLandingPageLocalDefault()],
    }),
    UserModule,
  ],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

Notice that the **UserModule** class has already been added in the AppModule file, this is how powerful the nest cli is.

Now all the basic things are finished and we can now run our project. To run the project enter the following command:

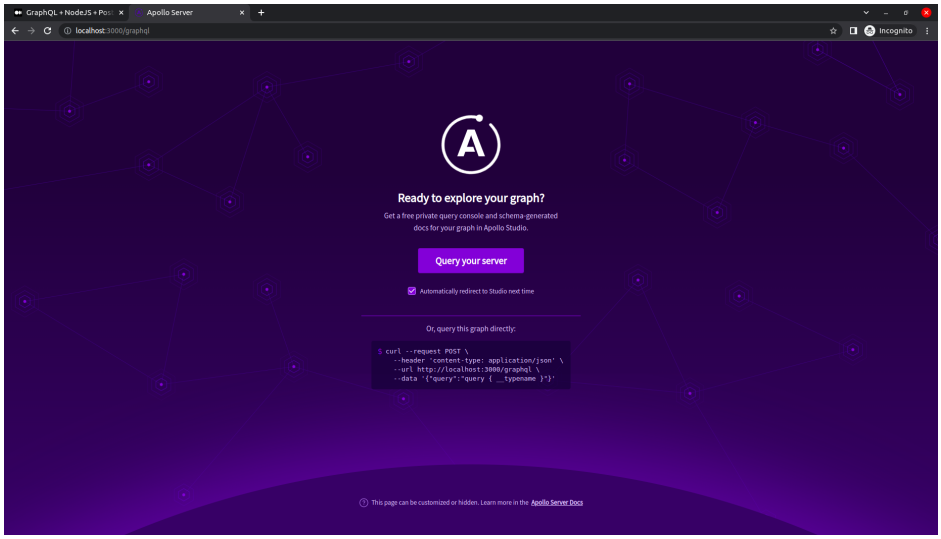
```
$ npm run start:dev
```

If everything is good then the app will run successfully and you should see the following message:

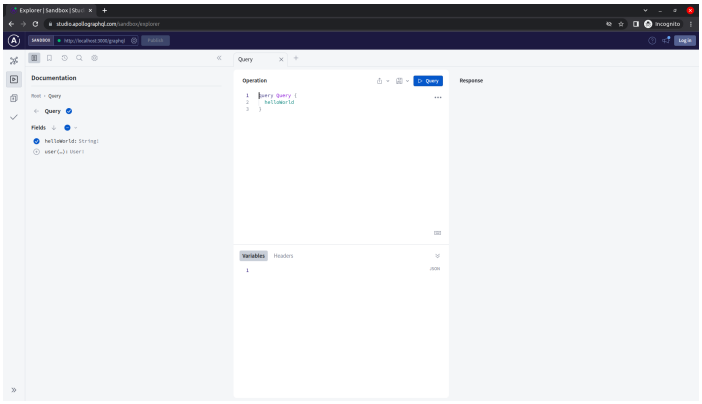
```
[15:15:35] Starting compilation in watch mode...

[15:15:37] Found 0 errors. Watching for file changes.

[Nest] 33385  - 06/02/2023, 15:15:38     LOG [NestFactory] Starting Nest application...
[Nest] 33385  - 06/02/2023, 15:15:38     LOG [InstanceLoader] UserModule dependencies initialized +16ms
[Nest] 33385  - 06/02/2023, 15:15:38     LOG [InstanceLoader] AppModule dependencies initialized +0ms
[Nest] 33385  - 06/02/2023, 15:15:38     LOG [InstanceLoader] GraphQLSchemaBuilderModule dependencies initialized +0ms
[Nest] 33385  - 06/02/2023, 15:15:38     LOG [InstanceLoader] GraphQLModule dependencies initialized +1ms
[Nest] 33385  - 06/02/2023, 15:15:38     LOG [RoutesResolver] AppController {/}: +2ms
[Nest] 33385  - 06/02/2023, 15:15:38     LOG [RouterExplorer] Mapped {/, GET} route +2ms
[Nest] 33385  - 06/02/2023, 15:15:38     LOG [GraphQLModule] Mapped {/graphql, POST} route +47ms
[Nest] 33385  - 06/02/2023, 15:15:38     LOG [NestApplication] Nest application successfully started +1ms
```

Now your project is successfully running on your localhost, so open any web browser, I'll be using the Chrome browser, and enter the following URL: http://localhost:3000/graphql

You should see the following screen. If not then please check whether you have added the following line **plugins: [ApolloServerPluginLandingPageLocalDefault()]** in your **app.module.ts** file.

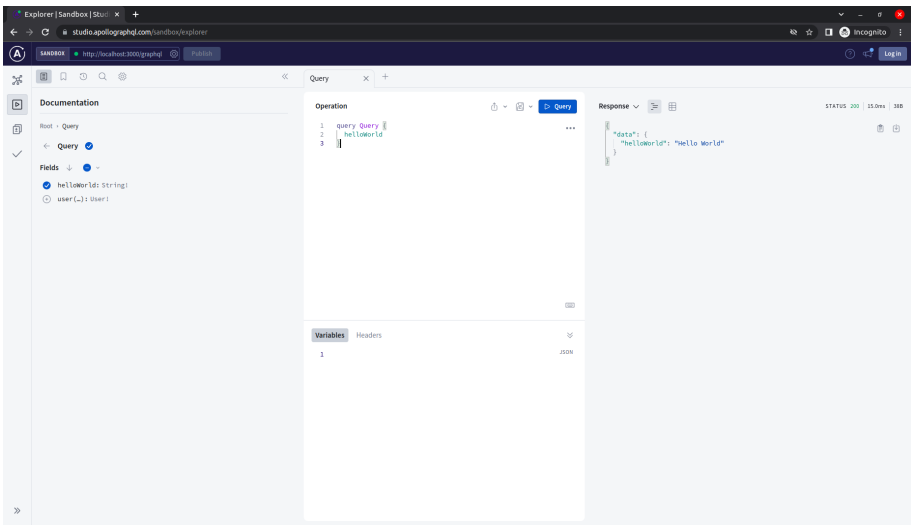


Check the box **“Automatically redirect to Studio next time”** and you should see the following screen:



Apollo Sandbox is a very powerful tool which is used to run our graphql queries without writing any queries manually.

Now let’s fire our first Query. On the left select the “**helloWorld: String!**” field and remove the id parameter from the query and click the big blue Example Query button (or press ctrl+enter) and voila! We have successfully fire our first GraphQL query from our nest project



Installing Postgres

Now our boilerplate is ready, let's add the Database. Here we will be using the famous postgres as our main Database. TypeORM is definitely the most mature Object Relational Mapper (ORM) available in the node.js world. Since it's written in TypeScript, it works pretty well with the Nest framework.

First we need to install postgres in our system.

```
$ npm install -g pg
```

Now setting up Postgres in our system is a very lengthy process, so I won't cover it in this article. You can install postgres in this url.

Once postgres is setup in your system, let's install TypeORM dependency. Run the following command:

```
$ npm i @nestjs/typeorm
```

Let's start by adding TypeORM into our **app.module.ts** file. Add the following code in the file:

```
TypeOrmModule.forRoot({
      type: 'postgres',
      database: 'postgres',
      entities: [],
      synchronize: true,
      host: 'localhost',
      port: 5432,
      username: 'postgres',
      password: 'postgres',
```

```
      autoLoadEntities: true,
    }),
```

Let's understand some of the import fields:

- **database**: The name of the database that we would be using.
- **synchronize**: Automatically generate schema on every application launch.
- **host**: 'localhost' by default.
- **port**: 5432 by default

app.module.ts:

```
import { ApolloDriver, ApolloDriverConfig } from '@nestjs/apollo';
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';
import { ApolloServerPluginLandingPageLocalDefault } from
'apollo-server-core';
import { join } from 'path';
import { TypeOrmModule } from '@nestjs/typeorm';

import { AppController } from './app.controller';
import { AppService } from './app.service';
import { UserModule } from './user/user.module';

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloDriverConfig>({
      driver: ApolloDriver,
      autoSchemaFile: join(process.cwd(), '/schema.gql'),
      sortSchema: true,
      playground: false,
      plugins: [ApolloServerPluginLandingPageLocalDefault()],
    }),
    TypeOrmModule.forRoot({
      type: 'postgres',
      database: 'postgres',
      entities: [],
      synchronize: true,
      host: 'localhost',
      port: 5432,
      username: 'postgres',
      password: 'postgres',
      autoLoadEntities: true,
    }),
    UserModule,
  ],
```

```
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

Now again run your application and if everything is good the app will run successfully.

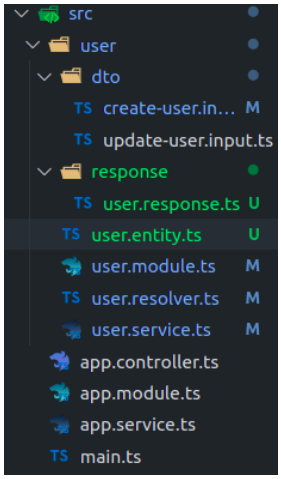
TypeORM and Postgres Queries

Now let's move towards the last and most important step in this article, postgres queries.

We would be following the repository pattern of TypeORM. So let's first do some modification in the naming of existing files. This is not mandatory step but I would like to follow the basic rules of naming so it won't be that confusing.

Let's rename the src/user/entity folder into src/user/response and inside that folder rename the **user.entity.ts** to **user.response.ts.**

Add a file named **user.entity.ts** in the **src/user** folder. Now the folder structure should look like this:



Now add the following to **user.entity.ts** file:

```
import { Entity, Column, PrimaryGeneratedColumn } from 'typeorm';

@Entity()
export class User {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  email: string;
```

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

This file would work as the schema for the User table. So now we need to add it to the TypeORM connection of our project, for that we need to add it inside the **UserModule** class.

user.module.ts

```
import { Module } from '@nestjs/common';
import { UserService } from './user.service';
import { UserResolver } from './user.resolver';
import { User } from './user.entity';
import { TypeOrmModule } from '@nestjs/typeorm';

@Module({
  imports: [TypeOrmModule.forFeature([User])],
  providers: [UserResolver, UserService],
})
export class UserModule {}
```

Now our Postgres DB is all setup and it will change the **user** table as per our schema/entity.

Let's create a simple createUser function which would add a new user entry in our **user** table. In GraphQL we have to define 2 things first:

- Input Values
- Responses

As we are using the **code first approach** we don't have to manually add it inside our **schema.gql** file. It will automatically get updated.

Add the following code inside **src/user/dto/create-user.input.ts** file:

create-user.input.ts:

```
import { InputType, Int, Field } from '@nestjs/graphql';

@InputType()
export class CreateUserInput {
  @Field()
  email: string;

  @Field()
```

```
  password: string;
}
```

Similarly add the following code in **src/user/response/user.response.ts** file

user.response.ts:

```
import { ObjectType, Field, Int } from '@nestjs/graphql';

@ObjectType()
export class User {
  @Field()
  email: string;

  @Field()
  password: string;
}
```

Perfect! Now our GraphQL schema is almost ready. Now let's add the final piece of code to run our first query, **createUser** service.

Add the following logic for creating user entry in our **user.service.ts** file.

user.service.ts:

```
constructor(@InjectRepository(User) private repo: Repository<User>) {}

async create(createUserInput: CreateUserInput) {
    const { email, password } = createUserInput;
    const user = this.repo.create({ email, password });

    return await this.repo.save(user);
}
```

In order to use the **Repository** pattern of **TypeORM** we have to Inject it inside our constructor of **UserService** class. After adding this our file should look like this:

user.service.ts:

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { CreateUserInput } from './dto/create-user.input';
import { UpdateUserInput } from './dto/update-user.input';
import { User } from './user.entity';

@Injectable()
```

```
export class UserService {
  constructor(@InjectRepository(User) private repo: Repository<User>) {}

  async create(createUserInput: CreateUserInput) {
    const { email, password } = createUserInput;
    const user = this.repo.create({ email, password });

    return await this.repo.save(user);
  }

  findAll() {
    return `This action returns all user`;
  }

  findOne(id: number) {
    return `This action returns a #${id} user`;
  }

  update(id: number, updateUserInput: UpdateUserInput) {
    return `This action updates a #${id} user`;
  }

  remove(id: number) {
    return `This action removes a #${id} user`;
  }
}
```

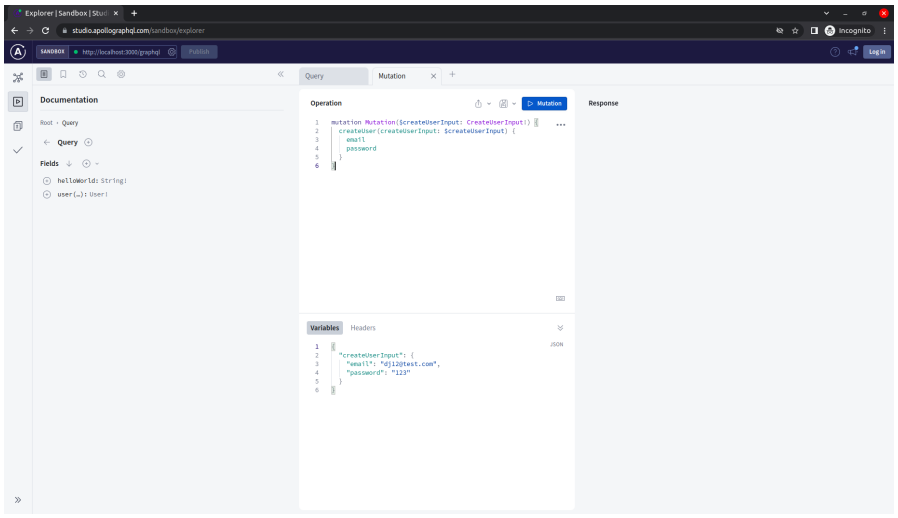
Now we are almost finished with our first service. Just make sure your **User Resolver** file should look like this:

user.resolver.ts:

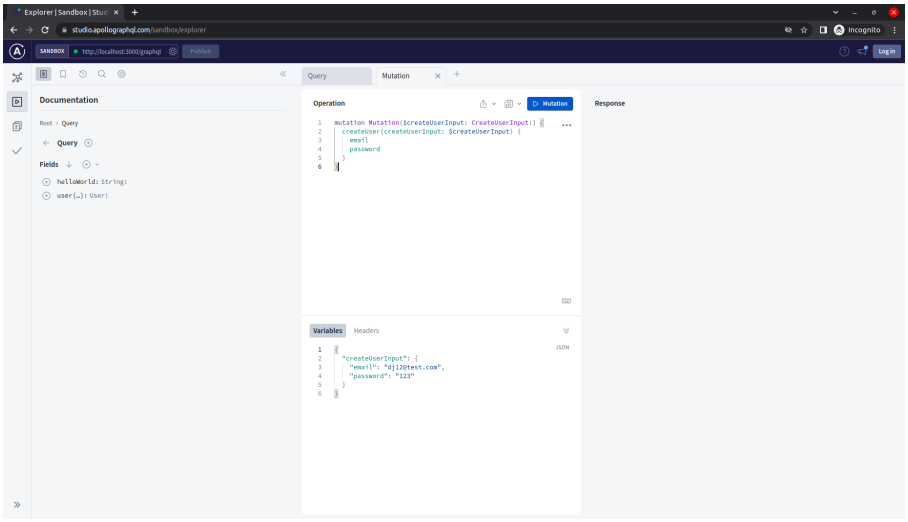
```
import { Resolver, Query, Mutation, Args, Int } from '@nestjs/graphql';
import { UserService } from './user.service';
import { User } from './response/user.response';
import { CreateUserInput } from './dto/create-user.input';
import { UpdateUserInput } from './dto/update-user.input';

@Resolver(() => User)
export class UserResolver {
  constructor(private readonly userService: UserService) {}

  @Mutation(() => User)
  createUser(@Args('createUserInput') createUserInput: CreateUserInput) {
    return this.userService.create(createUserInput);
```

```
  }

  @Query(() => [User], { name: 'user' })
  findAll() {
    return this.userService.findAll();
  }

  @Query(() => User, { name: 'user' })
  findOne(@Args('id', { type: () => Int }) id: number) {
    return this.userService.findOne(id);
  }

  @Mutation(() => User)
  updateUser(@Args('updateUserInput') updateUserInput: UpdateUserInput) {
    return this.userService.update(updateUserInput.id, updateUserInput);
  }

  @Mutation(() => User)
  removeUser(@Args('id', { type: () => Int }) id: number) {
    return this.userService.remove(id);
  }

  @Query((returns) => String)
  helloWorld() {
    return 'Hello World';
  }
}
```

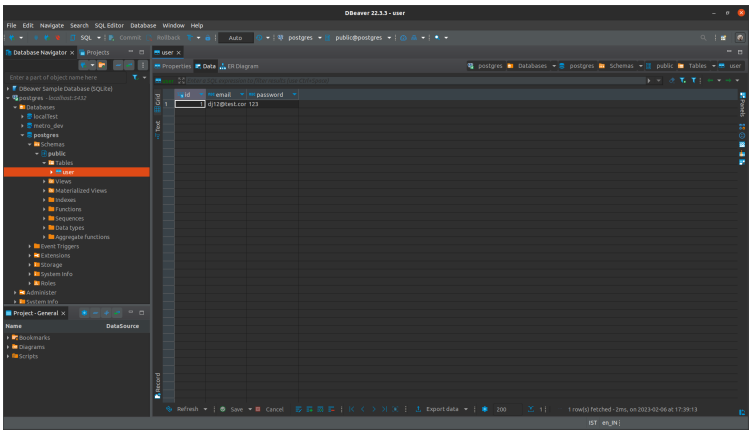
All Done! Let's run our application using the **npm run start:dev** command and it should run successfully. Now let's go back to our localhost url in the browser and fire the query:



Fire the query and voila! It ran successfully:



Now let's check in our Database Tool DBeaver and there should be a entry in our **user** table.



This was a simple project to understand the basic working of Nest.js with GraphQL using the **code first approach**.