Quick start

GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. It is an elegant approach that solves many of these problems that we have with typical REST apis. There is a great **comparison** between GraphQL and REST. In this set of articles, we're not going to explain what the GraphQL is, but rather show how to work with the dedicated `@nestjs/graphql` module. This chapter assumes that you are already familiar with GraphQL essentials.

The `GraphQLModule` is nothing more than a wrapper around the **Apollo** server. We don't reinvent the wheel but provide a ready to use module instead, that brings a clean way to play with the GraphQL and Nest together.

Installation

Firstly, we need to install the required packages:

```
$ npm i --save @nestjs/graphql apollo-server-express graphql-tools graphql
```

Overview

Nest offers two ways of building GraphQL applications, the schema first and the code first respectively.

In the **schema first** approach, the source of truth is a GraphQL SDL (Schema Definition Language). It's a language-agnostic way which basically allows you to share schema files between different platforms. Furthermore, Nest will automatically generate your TypeScript definitions based on the GraphQL schemas (using either classes or interfaces) to reduce redundancy.

In the **code first** approach on the other hand, you'll only use decorators and TypeScript classes to generate the corresponding GraphQL schema. It becomes very handy if you prefer to work exclusively with TypeScript and avoid the context switching between languages syntax.

Getting started

Once the packages are installed, we can register the `GraphQLModule`.

JS

```
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';

@Module({
  imports: [
    GraphQLModule.forRoot({}),
```

```
  ],
})
export class ApplicationModule {}
```

The `.forRoot()` method takes an options object as an argument. These options will be passed down to the underlying Apollo instance (read more about available settings **here**). For instance, if you want to disable the `playground` and turn off the `debug` mode, simply pass the following options:

JS

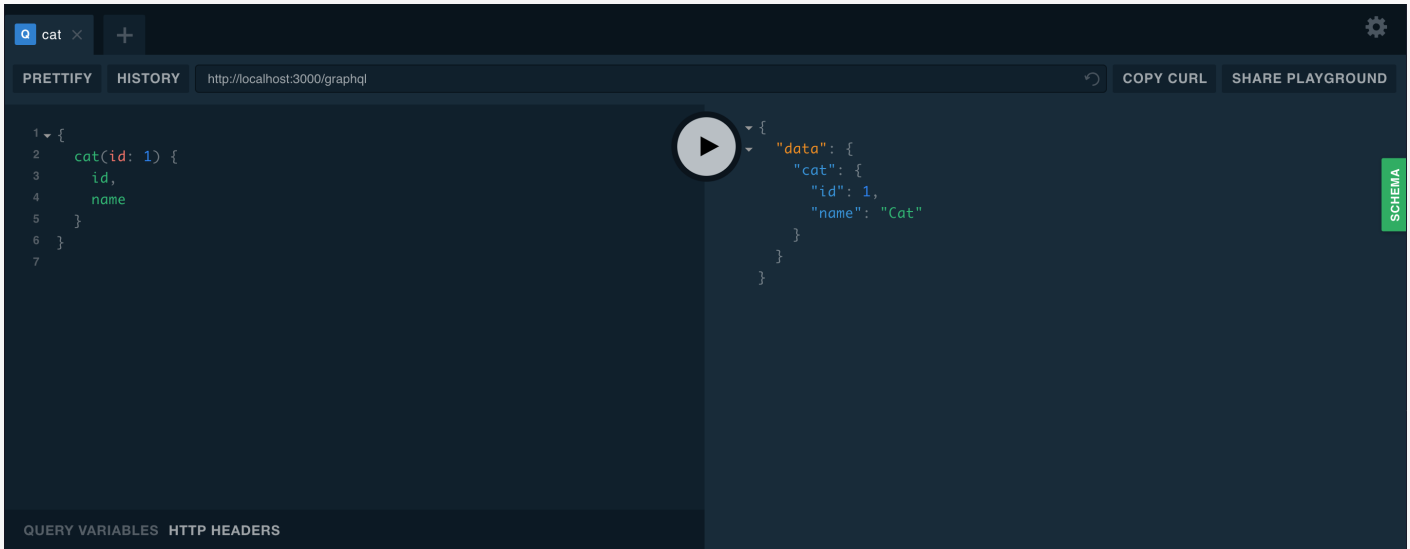
```
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';

@Module({
  imports: [
    GraphQLModule.forRoot({
      debug: false,
      playground: false,
    }),
  ],
})
export class ApplicationModule {}
```

As mentioned, all these settings will be forwarded to the `ApolloServer` constructor.

Playground

The playground is a graphical, interactive, in-browser GraphQL IDE, available by default on the same URL as the GraphQL server itself. Whilst your application is running in the background, open your web browser and navigate to `http://localhost:3000/graphql` (host and port may vary depending on your configuration).



Multiple endpoints

Another useful feature of this module is a capability to serve multiple endpoints at once. Thanks to that, you can decide which modules should be included in which endpoint. By default, `GraphQL` searches for resolvers throughout the whole app. To limit only a subset of modules, you can use the `include` property.

```
GraphQLModule.forRoot({
  include: [CatsModule],
}),
```

Schema first

To start using schema first way, simply add `typePaths` array inside the options object.

```
GraphQLModule.forRoot({
  typePaths: ['./**/*.graphql'],
}),
```

The `typePaths` property indicates where the `GraphQLModule` should look for the GraphQL files. All those files will be eventually combined in the memory which means that you can split your schemas into several files and hold them near to their resolvers.

Separate creation of both GraphQL types and corresponding TypeScript definitions creates unnecessary redundancy. Eventually, we end up without a single source of truth and each change made within SDL forces us to adjust interfaces as well. Thus, the `@nestjs/graphql` package serves another interesting functionality, which is the automatic generation of TS definitions using abstract syntax tree (AST). In order to enable it, simply add `definitions` property.

```
GraphQLModule.forRoot({
  typePaths: ['./**/*.graphql'],
  definitions: {
    path: join(process.cwd(), 'src/graphql.ts'),
  },
}),
```

The `src/graphql.ts` indicates where to save TypeScript output. By default, all types are transformed to the interfaces.

However, you can switch to classes instead by changing `outputAs` property to `class`.

```
GraphQLModule.forRoot({
  typePaths: ['./**/*.graphql'],
  definitions: {
    path: join(process.cwd(), 'src/graphql.ts'),
    outputAs: 'class',
  },
}),
```

However, generating type definitions on each application start may not be necessary. Instead, we might prefer to have full control, produce typings only when a dedicated command has been executed. In this case, we can create our own script, let's say `generate-typings.ts`:

```
import { GraphQLDefinitionsFactory } from '@nestjs/graphql';
import { join } from 'path';

const definitionsFactory = new GraphQLDefinitionsFactory();
definitionsFactory.generate({
  typePaths: ['./src/**/*.graphql'],
  path: join(process.cwd(), 'src/graphql.ts'),
  outputAs: 'class',
});
```

Afterward, simply run your file:

```
ts-node generate-typings
```

> **HINT**
> You can also compile a script beforehand and use `node` executable instead.

In order to switch to the watch mode (automatically generate typings on any `.graphql` file change), pass `watch` option to the `generate()` method.

```
definitionsFactory.generate({
  typePaths: ['./src/**/*.graphql'],
  path: join(process.cwd(), 'src/graphql.ts'),
```

```
  outputAs: 'class',
  watch: true,
});
```

A fully working sample is available **here**.

Code first

In the **code first** approach, you'll only use decorators and TypeScript classes to generate the corresponding GraphQL schema.

Nest is using an amazing **type-graphql** library under the hood in order provide this functionality. Hence, before we proceed, you have to install this package.

```
$ npm i type-graphql
```

Once the installation process is completed, we can add `autoSchemaFile` property to the options object.

```
GraphQLModule.forRoot({
  autoSchemaFile: 'schema.gql',
}),
```

The `autoSchemaFile` indicates a path where your automatically generated schema will be created. Additionally, you can pass the `buildSchemaOptions` property - an options object which will be passed in to the `buildSchema()` function (from the `type-graphql` package).

A fully working sample is available **here**.

Async configuration

Quite often you might want to asynchronously pass your module options instead of passing them beforehand. In such case, use `forRootAsync()` method, that provides a couple of various ways to deal with async data.

First possible approach is to use a factory function:

```
GraphQLModule.forRootAsync({
  useFactory: () => ({
    typePaths: ['./**/*.graphql'],
  }),
}),
```

Obviously, our factory behaves like every other one (might be `async` and is able to inject dependencies through `inject`).

```
GraphQLModule.forRootAsync({
  imports: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
    typePaths: configService.getString('GRAPHQL_TYPE_PATHS'),
  }),
  inject: [ConfigService],
}),
```

Alternatively, you are able to use class instead of a factory.

```
GraphQLModule.forRootAsync({
  useClass: GqlConfigService,
}),
```

Above construction will instantiate `GqlConfigService` inside `GraphQLModule` and will leverage it to create options object. The `GqlConfigService` has to implement `GqlOptionsFactory` interface.

```
@Injectable()
class GqlConfigService implements GqlOptionsFactory {
  createGqlOptions(): GqlModuleOptions {
    return {
      typePaths: ['./**/*.graphql'],
    };
  }
}
```

In order to prevent the creation of `GqlConfigService` inside `GraphQLModule` and use a provider imported from a different module, you can use the `useExisting` syntax.

```
GraphQLModule.forRootAsync({
  imports: [ConfigModule],
  useExisting: ConfigService,
```

```
});
```

It works the same as `useClass` with one critical difference - `GraphQLModule` will lookup imported modules to reuse already created `ConfigService`, instead of instantiating it on its own.

Support us

Nest is an MIT-licensed open source project. It can grow thanks to the support by these awesome people. If you'd like to join them, please read more here.

Principal Sponsor

Sponsors / Partners

Become a sponsor

Copyright © 2017-2019 MIT by Kamil Mysliwiec | design by Jakub Staron
Official NestJS Consulting Trilon.io | hosted by Netlify