

# Harnessing the power of TypeScript & GraphQL

[GraphQL](#) is a powerful query language for APIs and a runtime for fulfilling those queries with your existing data. It's an elegant approach that solves many problems typically found with REST APIs. For background, we suggest reading this [comparison](#) between GraphQL and REST. GraphQL combined with [TypeScript](#) helps you develop better type safety with your GraphQL queries, giving you end-to-end typing.

In this chapter, we assume a basic understanding of GraphQL, and focus on how to work with the built-in [@nestjs/graphql](#) module. The [GraphQLModule](#) can be configured to use [Apollo](#) server (with the [@nestjs/apollo](#) driver) and [Mercurius](#) (with the [@nestjs/mercurius](#)). We provide official integrations for these proven GraphQL packages to provide a simple way to use GraphQL with Nest (see more integrations [here](#)).

You can also build your own dedicated driver (read more on that [here](#)).

## Installation

Start by installing the required packages:

```
# For Express and Apollo (default)
$ npm i @nestjs/graphql @nestjs/apollo @apollo/server graphql

# For Fastify and Apollo
# npm i @nestjs/graphql @nestjs/apollo @apollo/server @as-
# integrations/fastify graphql

# For Fastify and Mercurius
# npm i @nestjs/graphql @nestjs/mercurius graphql mercurius
```

**Warning** [@nestjs/graphql@>=9](#) and [@nestjs/apollo^10](#) packages are compatible with **Apollo v3** (check out Apollo Server 3 [migration guide](#) for more details), while [@nestjs/graphql@^8](#) only supports **Apollo v2** (e.g., [apollo-server-express@2.x.x](#) package).

## Overview

Nest offers two ways of building GraphQL applications, the **code first** and the **schema first** methods. You should choose the one that works best for you. Most of the chapters in this GraphQL section are divided into two main parts: one you should follow if you adopt **code first**, and the other to be used if you adopt **schema first**.

In the **code first** approach, you use decorators and TypeScript classes to generate the corresponding GraphQL schema. This approach is useful if you prefer to work exclusively with TypeScript and avoid context switching between language syntaxes.

In the **schema first** approach, the source of truth is GraphQL SDL (Schema Definition Language) files. SDL is a language-agnostic way to share schema files between different platforms. Nest automatically generates

your TypeScript definitions (using either classes or interfaces) based on the GraphQL schemas to reduce the need to write redundant boilerplate code.

## Getting started with GraphQL & TypeScript

**info Hint** In the following chapters, we'll be integrating the `@nestjs/apollo` package. If you want to use `mercurius` package instead, navigate to [this section](#).

Once the packages are installed, we can import the `GraphQLModule` and configure it with the `forRoot()` static method.

```
@@filename()  
import { Module } from '@nestjs/common';  
import { GraphQLModule } from '@nestjs/graphql';  
import { ApolloDriver, ApolloDriverConfig } from '@nestjs/apollo';  
  
@Module({  
  imports: [  
    GraphQLModule.forRoot<ApolloDriverConfig>({  
      driver: ApolloDriver,  
    }),  
  ],  
})  
export class AppModule {}
```

**info Hint** For `mercurius` integration, you should be using the `MercuriusDriver` and `MercuriusDriverConfig` instead. Both are exported from the `@nestjs/mercurius` package.

The `forRoot()` method takes an options object as an argument. These options are passed through to the underlying driver instance (read more about available settings here: [Apollo](#) and [Mercurius](#)). For example, if you want to disable the `playground` and turn off `debug` mode (for Apollo), pass the following options:

```
@@filename()  
import { Module } from '@nestjs/common';  
import { GraphQLModule } from '@nestjs/graphql';  
import { ApolloDriver, ApolloDriverConfig } from '@nestjs/apollo';  
  
@Module({  
  imports: [  
    GraphQLModule.forRoot<ApolloDriverConfig>({  
      driver: ApolloDriver,  
      playground: false,  
    }),  
  ],  
})  
export class AppModule {}
```

In this case, these options will be forwarded to the `ApolloServer` constructor.

## GraphQL playground

The playground is a graphical, interactive, in-browser GraphQL IDE, available by default on the same URL as the GraphQL server itself. To access the playground, you need a basic GraphQL server configured and running. To see it now, you can install and build the [working example here](#). Alternatively, if you're following along with these code samples, once you've completed the steps in the [Resolvers chapter](#), you can access the playground.

With that in place, and with your application running in the background, you can then open your web browser and navigate to <http://localhost:3000/graphql> (host and port may vary depending on your configuration). You will then see the GraphQL playground, as shown below.

warning **Note** `@nestjs/mercurius` integration does not ship with the built-in GraphQL Playground integration. Instead, you can use `GraphiQL` (set `graphiql: true`).

## Multiple endpoints

Another useful feature of the `@nestjs/graphql` module is the ability to serve multiple endpoints at once. This lets you decide which modules should be included in which endpoint. By default, `GraphQL` searches for resolvers throughout the whole app. To limit this scan to only a subset of modules, use the `include` property.

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

warning **Warning** If you use the `@apollo/server` with `@as-integrations/fastify` package with multiple GraphQL endpoints in a single application, make sure to enable the `disableHealthCheck` setting in the `GraphQLModule` configuration.

## Code first

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

To use the code first approach, start by adding the `autoSchemaFile` property to the options object:

```
GraphQLModule.forRoot<ApolloDriverConfig>({  
  driver: ApolloDriver,  
  autoSchemaFile: join(process.cwd(), 'src/schema.gql'),  
}),
```

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`:

```
GraphQLModule.forRoot<ApolloDriverConfig>({  
  driver: ApolloDriver,  
  autoSchemaFile: 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`:

```
GraphQLModule.forRoot<ApolloDriverConfig>({  
  driver: ApolloDriver,  
  autoSchemaFile: join(process.cwd(), 'src/schema.gql'),  
  sortSchema: true,  
}),
```

## Example

A fully working code first sample is available [here](#).

## Schema first

To use the schema first approach, start by adding a `typePaths` property to the options object. The `typePaths` property indicates where the `GraphQLModule` should look for GraphQL SDL schema definition files you'll be writing. These files will be combined in memory; this allows you to split your schemas into several files and locate them near their resolvers.

```
GraphQLModule.forRoot<ApolloDriverConfig>({  
  driver: ApolloDriver,  
  typePaths: ['./**/*.graphql'],  
}),
```

You will typically also need to have TypeScript definitions (classes and interfaces) that correspond to the GraphQL SDL types. Creating the corresponding TypeScript definitions by hand is redundant and tedious. It leaves us without a single source of truth -- each change made within SDL forces us to adjust TypeScript definitions as well. To address this, the `@nestjs/graphql` package can **automatically generate** TypeScript definitions from the abstract syntax tree (AST). To enable this feature, add the `definitions` options property when configuring the `GraphQLModule`.

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

The path property of the `definitions` object indicates where to save generated TypeScript output. By default, all generated TypeScript types are created as interfaces. To generate classes instead, specify the `outputAs` property with a value of `'class'`.

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

The above approach dynamically generates TypeScript definitions each time the application starts. Alternatively, it may be preferable to build a simple script to generate these on demand. For example, assume we create the following script as `generate-typings.ts`:

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

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

Now you can run this script on demand:

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

**info Hint** You can compile the script beforehand (e.g., with `tsc`) and use `node` to execute it.

To enable watch mode for the script (to automatically generate typings whenever any `.graphql` file changes), pass the `watch` option to the `generate()` method.

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

To automatically generate the additional `__typename` field for every object type, enable the `emitTypenameField` option.

```
definitionsFactory.generate({
  // ...,
  emitTypenameField: true,
});
```

To generate resolvers (queries, mutations, subscriptions) as plain fields without arguments, enable the `skipResolverArgs` option.

```
definitionsFactory.generate({
  // ...,
  skipResolverArgs: true,
});
```

## Apollo Sandbox

To use [Apollo Sandbox](#) instead of the `graphql-playground` as a GraphQL IDE for local development, use the following configuration:

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

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloDriverConfig>({
      driver: ApolloDriver,
      playground: false,
      plugins: [ApolloServerPluginLandingPageLocalDefault()],
    }),
  ],
})
export class AppModule {}
```

## Example

A fully working schema first sample is available [here](#).

## Accessing generated schema

In some circumstances (for example end-to-end tests), you may want to get a reference to the generated schema object. In end-to-end tests, you can then run queries using the `graphql` object without using any HTTP listeners.

You can access the generated schema (in either the code first or schema first approach), using the `GraphQLSchemaHost` class:

```
const { schema } = app.get(GraphQLSchemaHost);
```

**info Hint** You must call the `GraphQLSchemaHost#schema` getter after the application has been initialized (after the `onModuleInit` hook has been triggered by either the `app.listen()` or `app.init()` method).

## Async configuration

When you need to pass module options asynchronously instead of statically, use the `forRootAsync()` method. As with most dynamic modules, Nest provides several techniques to deal with async configuration.

One technique is to use a factory function:

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

Like other factory providers, our factory function can be `async` and can inject dependencies through `inject`.

```
GraphQLModule.forRootAsync<ApolloDriverConfig>({
  driver: ApolloDriver,
  imports: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
    typePaths: configService.get<string>('GRAPHQL_TYPE_PATHS'),
  }),
  inject: [ConfigService],
}),
```

Alternatively, you can configure the `GraphQLModule` using a class instead of a factory, as shown below:

```
GraphQLModule.forRootAsync<ApolloDriverConfig>({
  driver: ApolloDriver,
  useClass: GqlConfigService,
}),
```

The construction above instantiates `GqlConfigService` inside `GraphQLModule`, using it to create options object. Note that in this example, the `GqlConfigService` has to implement the `GqlOptionsFactory` interface, as shown below. The `GraphQLModule` will call the `createGqlOptions()` method on the instantiated object of the supplied class.

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

If you want to reuse an existing options provider instead of creating a private copy inside the `GraphQLModule`, use the `useExisting` syntax.

```
GraphQLModule.forRootAsync<ApolloDriverConfig>({
  imports: [ConfigModule],
  useExisting: ConfigService,
}),
```

## Mercurius integration

Instead of using Apollo, Fastify users (read more [here](#)) can alternatively use the `@nestjs/mercurius` driver.

```
@@filename()
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';
import { MercuriusDriver, MercuriusDriverConfig } from '@nestjs/mercurius';

@Module({
  imports: [
    GraphQLModule.forRoot<MercuriusDriverConfig>({
      driver: MercuriusDriver,
      graphiql: true,
    }),
  ],
})
export class AppModule {}
```



info **Hint** Once the application is running, open your browser and navigate to <http://localhost:3000/graphiql>. You should see the [GraphQL IDE](#).

The `forRoot()` method takes an options object as an argument. These options are passed through to the underlying driver instance. Read more about available settings [here](#).

### Third-party integrations

- [GraphQL Yoga](#)

### Example

A working example is available [here](#).