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 **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 '@nestjs/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.

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. To demonstrate the process of using the package features to create a GraphQL API, we'll create a simple authors API.

Code first

In the code first approach, we don't follow the typical process of creating our GraphQL schema by writing GraphQL SDL by hand. Instead, we use TypeScript decorators to generate the SDL from TypeScript class definitions. The `@nestjs/graphql` package reads the metadata defined through the decorators and automatically generates the schema for you.

Object types

Most of the definitions in a GraphQL schema are **object types**. Each object type you define should represent a domain object that an application client might need to interact with. For example, our sample API needs to be able to fetch a list of authors and their posts, so we should define the `Author` type and `Post` type to support this functionality.

If we were using the schema first approach, we'd define such a schema with SDL like this:

```
type Author {
  id: Int!
  firstName: String
  lastName: String
  posts: [Post!]!
}
```

In this case, using the code first approach, we define schemas using TypeScript classes and using TypeScript decorators to annotate the fields of those classes. The equivalent of the above SDL in the code first approach is:

```
@@filename(authors/models/author.model)
import { Field, Int, ObjectType } from '@nestjs/graphql';
import { Post } from './post';

@ObjectType()
export class Author {
  @Field(type => Int)
  id: number;

  @Field({ nullable: true })
  firstName?: string;
```

```
  @Field({ nullable: true })
  lastName?: string;

  @Field(type => [Post])
  posts: Post[];
}
```

> info **Hint** TypeScript's metadata reflection system has several limitations which make it impossible, for instance, to determine what properties a class consists of or recognize whether a given property is optional or required. Because of these limitations, we must either explicitly use the `@Field()` decorator in our schema definition classes to provide metadata about each field's GraphQL type and optionality, or use a CLI plugin to generate these for us.

The `Author` object type, like any class, is made of a collection of fields, with each field declaring a type. A field's type corresponds to a GraphQL type. A field's GraphQL type can be either another object type or a scalar type. A GraphQL scalar type is a primitive (like `ID`, `String`, `Boolean`, or `Int`) that resolves to a single value.

> info **Hint** In addition to GraphQL's built-in scalar types, you can define custom scalar types (read more).

The above `Author` object type definition will cause Nest to **generate** the SDL we showed above:

```
type Author {
  id: Int!
  firstName: String
  lastName: String
  posts: [Post!]!
}
```

The `@Field()` decorator accepts an optional type function (e.g., `type => Int`), and optionally an options object.

The type function is required when there's the potential for ambiguity between the TypeScript type system and the GraphQL type system. Specifically: it is **not** required for `string` and `boolean` types; it **is** required for `number` (which must be mapped to either a GraphQL `Int` or `Float`). The type function should simply return the desired GraphQL type (as shown in various examples in these chapters).

The options object can have any of the following key/value pairs:

- `nullable`: for specifying whether a field is nullable (in SDL, each field is non-nullable by default); `boolean`
- `description`: for setting a field description; `string`
- `deprecationReason`: for marking a field as deprecated; `string`

For example:

```
@Field({ description: `Book title`, deprecationReason: 'Not useful in v2
schema' })
title: string;
```

> info **Hint** You can also add a description to, or deprecate, the whole object type: `@ObjectType({{ '{' }} description: 'Author model' {{ '}' }})`.

When the field is an array, we must manually indicate the array type in the `Field()` decorator's type function, as shown below:

```
@Field(type => [Post])
posts: Post[];
```

> info **Hint** Using array bracket notation (`[ ]`), we can indicate the depth of the array. For example, using `[[Int]]` would represent an integer matrix.

To declare that an array's items (not the array itself) are nullable, set the `nullable` property to `'items'` as shown below:

```
@Field(type => [Post], { nullable: 'items' })
posts: Post[];
```

> info **Hint** If both the array and its items are nullable, set `nullable` to `'itemsAndList'` instead.

Now that the `Author` object type is created, let's define the `Post` object type.

```
@@filename(posts/models/post.model)
import { Field, Int, ObjectType } from '@nestjs/graphql';

@ObjectType()
export class Post {
  @Field(type => Int)
  id: number;

  @Field()
  title: string;

  @Field(type => Int, { nullable: true })
  votes?: number;
}
```

The `Post` object type will result in generating the following part of the GraphQL schema in SDL:

```
type Post {
  id: Int!
  title: String!
  votes: Int
}
```

Code first resolver

At this point, we've defined the objects (type definitions) that can exist in our data graph, but clients don't yet have a way to interact with those objects. To address that, we need to create a resolver class. In the code first method, a resolver class both defines resolver functions **and** generates the **Query type**. This will be clear as we work through the example below:

```
@@filename(authors/authors.resolver)
@Resolver(of => Author)
export class AuthorsResolver {
  constructor(
    private authorsService: AuthorsService,
    private postsService: PostsService,
  ) {}

  @Query(returns => Author)
  async author(@Args('id', { type: () => Int }) id: number) {
    return this.authorsService.findOneById(id);
  }

  @ResolveField()
  async posts(@Parent() author: Author) {
    const { id } = author;
    return this.postsService.findAll({ authorId: id });
  }
}
```

> info **Hint** All decorators (e.g., `@Resolver`, `@ResolveField`, `@Args`, etc.) are exported from the `@nestjs/graphql` package.

You can define multiple resolver classes. Nest will combine these at run time. See the module section below for more on code organization.

> warning **Note** The logic inside the `AuthorsService` and `PostsService` classes can be as simple or sophisticated as needed. The main point of this example is to show how to construct resolvers and how they can interact with other providers.

In the example above, we created the `AuthorsResolver` which defines one query resolver function and one field resolver function. To create a resolver, we create a class with resolver functions as methods, and annotate the class with the `@Resolver()` decorator.

In this example, we defined a query handler to get the author object based on the `id` sent in the request. To specify that the method is a query handler, use the `@Query()` decorator.

The argument passed to the `@Resolver()` decorator is optional, but comes into play when our graph becomes non-trivial. It's used to supply a parent object used by field resolver functions as they traverse down through an object graph.

In our example, since the class includes a **field resolver** function (for the `posts` property of the `Author` object type), we **must** supply the `@Resolver()` decorator with a value to indicate which class is the parent type (i.e., the corresponding `ObjectType` class name) for all field resolvers defined within this class. As should be clear from the example, when writing a field resolver function, it's necessary to access the parent object (the object the field being resolved is a member of). In this example, we populate an author's posts array with a field resolver that calls a service which takes the author's `id` as an argument. Hence the need to identify the parent object in the `@Resolver()` decorator. Note the corresponding use of the `@Parent()` method parameter decorator to then extract a reference to that parent object in the field resolver.

We can define multiple `@Query()` resolver functions (both within this class, and in any other resolver class), and they will be aggregated into a single **Query type** definition in the generated SDL along with the appropriate entries in the resolver map. This allows you to define queries close to the models and services that they use, and to keep them well organized in modules.

> info **Hint** Nest CLI provides a generator (schematic) that automatically generates **all the boilerplate code** to help us avoid doing all of this, and make the developer experience much simpler. Read more about this feature here.

Query type names

In the above examples, the `@Query()` decorator generates a GraphQL schema query type name based on the method name. For example, consider the following construction from the example above:

```
@Query(returns => Author)
async author(@Args('id', { type: () => Int }) id: number) {
  return this.authorsService.findOneById(id);
}
```

This generates the following entry for the author query in our schema (the query type uses the same name as the method name):

```
type Query {
  author(id: Int!): Author
}
```

> info **Hint** Learn more about GraphQL queries here.

Conventionally, we prefer to decouple these names; for example, we prefer to use a name like `getAuthor()` for our query handler method, but still use `author` for our query type name. The same

applies to our field resolvers. We can easily do this by passing the mapping names as arguments of the `@Query()` and `@ResolveField()` decorators, as shown below:

```
@@filename(authors/authors.resolver)
@Resolver(of => Author)
export class AuthorsResolver {
  constructor(
    private authorsService: AuthorsService,
    private postsService: PostsService,
  ) {}

  @Query(returns => Author, { name: 'author' })
  async getAuthor(@Args('id', { type: () => Int }) id: number) {
    return this.authorsService.findOneById(id);
  }

  @ResolveField('posts', returns => [Post])
  async getPosts(@Parent() author: Author) {
    const { id } = author;
    return this.postsService.findAll({ authorId: id });
  }
}
```

The `getAuthor` handler method above will result in generating the following part of the GraphQL schema in SDL:

```
type Query {
  author(id: Int!): Author
}
```

Query decorator options

The `@Query()` decorator's options object (where we pass `{{ '{' }}name: 'author'{{ '}' }}` above) accepts a number of key/value pairs:

- `name`: name of the query; a `string`
- `description`: a description that will be used to generate GraphQL schema documentation (e.g., in GraphQL playground); a `string`
- `deprecationReason`: sets query metadata to show the query as deprecated (e.g., in GraphQL playground); a `string`
- `nullable`: whether the query can return a null data response; `boolean` or `'items'` or `'itemsAndList'` (see above for details of `'items'` and `'itemsAndList'`)

Args decorator options

Use the `@Args()` decorator to extract arguments from a request for use in the method handler. This works in a very similar fashion to REST route parameter argument extraction.

Usually your `@Args()` decorator will be simple, and not require an object argument as seen with the `getAuthor()` method above. For example, if the type of an identifier is string, the following construction is sufficient, and simply plucks the named field from the inbound GraphQL request for use as a method argument.

```
@Args('id') id: string
```

In the `getAuthor()` case, the `number` type is used, which presents a challenge. The `number` TypeScript type doesn't give us enough information about the expected GraphQL representation (e.g., `Int` vs. `Float`). Thus we have to **explicitly** pass the type reference. We do that by passing a second argument to the `Args()` decorator, containing argument options, as shown below:

```
@Query(returns => Author, { name: 'author' })
async getAuthor(@Args('id', { type: () => Int }) id: number) {
  return this.authorsService.findOneById(id);
}
```

The options object allows us to specify the following optional key value pairs:

- `type`: a function returning the GraphQL type
- `defaultValue`: a default value; `any`
- `description`: description metadata; `string`
- `deprecationReason`: to deprecate a field and provide meta data describing why; `string`
- `nullable`: whether the field is nullable

Query handler methods can take multiple arguments. Let's imagine that we want to fetch an author based on its `firstName` and `lastName`. In this case, we can call `@Args` twice:

```
getAuthor(
  @Args('firstName', { nullable: true }) firstName?: string,
  @Args('lastName', { defaultValue: '' }) lastName?: string,
) {}
```

Dedicated arguments class

With inline `@Args()` calls, code like the example above becomes bloated. Instead, you can create a dedicated `GetAuthorArgs` arguments class and access it in the handler method as follows:

```
@Args() args: GetAuthorArgs
```

Create the `GetAuthorArgs` class using `@ArgsType()` as shown below:

```
@@filename(authors/dto/get-author.args)
import { MinLength } from 'class-validator';
import { Field, ArgsType } from '@nestjs/graphql';

@ArgsType()
class GetAuthorArgs {
  @Field({ nullable: true })
  firstName?: string;

  @Field({ defaultValue: '' })
  @MinLength(3)
  lastName: string;
}
```

> info **Hint** Again, due to TypeScript's metadata reflection system limitations, it's required to either use the `@Field` decorator to manually indicate type and optionality, or use a CLI plugin.

This will result in generating the following part of the GraphQL schema in SDL:

```
type Query {
  author(firstName: String, lastName: String = ''): Author
}
```

> info **Hint** Note that arguments classes like `GetAuthorArgs` play very well with the `ValidationPipe` (read more).

Class inheritance

You can use standard TypeScript class inheritance to create base classes with generic utility type features (fields and field properties, validations, etc.) that can be extended. For example, you may have a set of pagination related arguments that always include the standard `offset` and `limit` fields, but also other index fields that are type-specific. You can set up a class hierarchy as shown below.

Base `@ArgsType()` class:

```
@ArgsType()
class PaginationArgs {
  @Field((type) => Int)
  offset: number = 0;

  @Field((type) => Int)
  limit: number = 10;
}
```

Type specific sub-class of the base `@ArgsType()` class:

```
@ArgsType()
class GetAuthorArgs extends PaginationArgs {
  @Field({ nullable: true })
  firstName?: string;

  @Field({ defaultValue: '' })
  @MinLength(3)
  lastName: string;
}
```

The same approach can be taken with `@ObjectType()` objects. Define generic properties on the base class:

```
@ObjectType()
class Character {
  @Field((type) => Int)
  id: number;

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

Add type-specific properties on sub-classes:

```
@ObjectType()
class Warrior extends Character {
  @Field()
  level: number;
}
```

You can use inheritance with a resolver as well. You can ensure type safety by combining inheritance and TypeScript generics. For example, to create a base class with a generic `findAll` query, use a construction like this:

```
function BaseResolver<T extends Type<unknown>>(classRef: T): any {
  @Resolver({ isAbstract: true })
  abstract class BaseResolverHost {
    @Query((type) => [classRef], { name: `findAll${classRef.name}` })
    async findAll(): Promise<T[]> {
      return [];
    }
  }
  return BaseResolverHost;
}
```

Note the following:

- an explicit return type (`any` above) is required: otherwise TypeScript complains about the usage of a private class definition. Recommended: define an interface instead of using `any`.
- `Type` is imported from the `@nestjs/common` package
- The `isAbstract: true` property indicates that SDL (Schema Definition Language statements) shouldn't be generated for this class. Note, you can set this property for other types as well to suppress SDL generation.

Here's how you could generate a concrete sub-class of the `BaseResolver`:

```
@Resolver((of) => Recipe)
export class RecipesResolver extends BaseResolver(Recipe) {
  constructor(private recipesService: RecipesService) {
    super();
  }
}
```

This construct would generated the following SDL:

```
type Query {
  findAllRecipe: [Recipe!]!
}
```

Generics

We saw one use of generics above. This powerful TypeScript feature can be used to create useful abstractions. For example, here's a sample cursor-based pagination implementation based on this documentation:

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

interface IEdgeType<T> {
  cursor: string;
  node: T;
}

export interface IPaginatedType<T> {
  edges: IEdgeType<T>[];
  nodes: T[];
  totalCount: number;
  hasNextPage: boolean;
}

export function Paginated<T>(classRef: Type<T>): Type<IPaginatedType<T>> {
  @ObjectType(`${classRef.name}Edge`)
```

```
  abstract class EdgeType {
    @Field((type) => String)
    cursor: string;

    @Field((type) => classRef)
    node: T;
  }

  @ObjectType({ isAbstract: true })
  abstract class PaginatedType implements IPaginatedType<T> {
    @Field((type) => [EdgeType], { nullable: true })
    edges: EdgeType[];

    @Field((type) => [classRef], { nullable: true })
    nodes: T[];

    @Field((type) => Int)
    totalCount: number;

    @Field()
    hasNextPage: boolean;
  }
  return PaginatedType as Type<IPaginatedType<T>>;
}
```

With the above base class defined, we can now easily create specialized types that inherit this behavior. For example:

```
@ObjectType()
class PaginatedAuthor extends Paginated(Author) {}
```

Schema first

As mentioned in the previous chapter, in the schema first approach we start by manually defining schema types in SDL (read more). Consider the following SDL type definitions.

> info **Hint** For convenience in this chapter, we've aggregated all of the SDL in one location (e.g., one `.graphql` file, as shown below). In practice, you may find it appropriate to organize your code in a modular fashion. For example, it can be helpful to create individual SDL files with type definitions representing each domain entity, along with related services, resolver code, and the Nest module definition class, in a dedicated directory for that entity. Nest will aggregate all the individual schema type definitions at run time.

```
type Author {
  id: Int!
  firstName: String
  lastName: String
  posts: [Post]
```

```
}

type Post {
  id: Int!
  title: String!
  votes: Int
}

type Query {
  author(id: Int!): Author
}
```

Schema first resolver

The schema above exposes a single query - `author(id: Int!): Author`.

> info **Hint** Learn more about GraphQL queries here.

Let's now create an `AuthorsResolver` class that resolves author queries:

```
@@filename(authors/authors.resolver)
@Resolver('Author')
export class AuthorsResolver {
  constructor(
    private authorsService: AuthorsService,
    private postsService: PostsService,
  ) {}

  @Query()
  async author(@Args('id') id: number) {
    return this.authorsService.findOneById(id);
  }

  @ResolveField()
  async posts(@Parent() author) {
    const { id } = author;
    return this.postsService.findAll({ authorId: id });
  }
}
```

> info **Hint** All decorators (e.g., `@Resolver`, `@ResolveField`, `@Args`, etc.) are exported from the `@nestjs/graphql` package.

> warning **Note** The logic inside the `AuthorsService` and `PostsService` classes can be as simple or sophisticated as needed. The main point of this example is to show how to construct resolvers and how they can interact with other providers.

The `@Resolver()` decorator is required. It takes an optional string argument with the name of a class. This class name is required whenever the class includes `@ResolveField()` decorators to inform Nest that the

decorated method is associated with a parent type (the `Author` type in our current example). Alternatively, instead of setting `@Resolver()` at the top of the class, this can be done for each method:

```
@Resolver('Author')
@ResolveField()
async posts(@Parent() author) {
  const { id } = author;
  return this.postsService.findAll({ authorId: id });
}
```

In this case (`@Resolver()` decorator at the method level), if you have multiple `@ResolveField()` decorators inside a class, you must add `@Resolver()` to all of them. This is not considered the best practice (as it creates extra overhead).

> info **Hint** Any class name argument passed to `@Resolver()` **does not** affect queries (`@Query()` decorator) or mutations (`@Mutation()` decorator).

> warning **Warning** Using the `@Resolver` decorator at the method level is not supported with the **code first** approach.

In the above examples, the `@Query()` and `@ResolveField()` decorators are associated with GraphQL schema types based on the method name. For example, consider the following construction from the example above:

```
@Query()
async author(@Args('id') id: number) {
  return this.authorsService.findOneById(id);
}
```

This generates the following entry for the author query in our schema (the query type uses the same name as the method name):

```
type Query {
  author(id: Int!): Author
}
```

Conventionally, we would prefer to decouple these, using names like `getAuthor()` or `getPosts()` for our resolver methods. We can easily do this by passing the mapping name as an argument to the decorator, as shown below:

```
@@filename(authors/authors.resolver)
@Resolver('Author')
export class AuthorsResolver {
  constructor(
    private authorsService: AuthorsService,
```

```
    private postsService: PostsService,
  ) {}

  @Query('author')
  async getAuthor(@Args('id') id: number) {
    return this.authorsService.findOneById(id);
  }

  @ResolveField('posts')
  async getPosts(@Parent() author) {
    const { id } = author;
    return this.postsService.findAll({ authorId: id });
  }
}
```

> info **Hint** Nest CLI provides a generator (schematic) that automatically generates **all the boilerplate code** to help us avoid doing all of this, and make the developer experience much simpler. Read more about this feature here.

Generating types

Assuming that we use the schema first approach and have enabled the typings generation feature (with `outputAs: 'class'` as shown in the previous chapter), once you run the application it will generate the following file (in the location you specified in the `GraphQLModule.forRoot()` method). For example, in `src/graphql.ts`:

```
@@filename(graphql)
export (class Author {
  id: number;
  firstName?: string;
  lastName?: string;
  posts?: Post[];
})
export class Post {
  id: number;
  title: string;
  votes?: number;
}

export abstract class IQuery {
  abstract author(id: number): Author | Promise<Author>;
}
```

By generating classes (instead of the default technique of generating interfaces), you can use declarative validation **decorators** in combination with the schema first approach, which is an extremely useful technique (read more). For example, you could add `class-validator` decorators to the generated `CreatePostInput` class as shown below to enforce minimum and maximum string lengths on the `title` field:

```
import { MinLength, MaxLength } from 'class-validator';

export class CreatePostInput {
  @MinLength(3)
  @MaxLength(50)
  title: string;
}
```

> warning **Notice** To enable auto-validation of your inputs (and parameters), use `ValidationPipe`. Read more about validation here and more specifically about pipes here.

However, if you add decorators directly to the automatically generated file, they will be **overwritten** each time the file is generated. Instead, create a separate file and simply extend the generated class.

```
import { MinLength, MaxLength } from 'class-validator';
import { Post } from '../../graphql.ts';

export class CreatePostInput extends Post {
  @MinLength(3)
  @MaxLength(50)
  title: string;
}
```

GraphQL argument decorators

We can access the standard GraphQL resolver arguments using dedicated decorators. Below is a comparison of the Nest decorators and the plain Apollo parameters they represent.

`@Root()` and `@Parent()`	`root`/`parent`
`@Context(param?: string)`	`context` / `context[param]`
`@Info(param?: string)`	`info` / `info[param]`
`@Args(param?: string)`	`args` / `args[param]`

These arguments have the following meanings:

- `root`: an object that contains the result returned from the resolver on the parent field, or, in the case of a top-level `Query` field, the `rootValue` passed from the server configuration.
- `context`: an object shared by all resolvers in a particular query; typically used to contain per-request state.
- `info`: an object that contains information about the execution state of the query.
- `args`: an object with the arguments passed into the field in the query.

Module

Once we're done with the above steps, we have declaratively specified all the information needed by the `GraphQLModule` to generate a resolver map. The `GraphQLModule` uses reflection to introspect the meta data provided via the decorators, and transforms classes into the correct resolver map automatically.

The only other thing you need to take care of is to **provide** (i.e., list as a `provider` in some module) the resolver class(es) (`AuthorsResolver`), and importing the module (`AuthorsModule`) somewhere, so Nest will be able to utilize it.

For example, we can do this in an `AuthorsModule`, which can also provide other services needed in this context. Be sure to import `AuthorsModule` somewhere (e.g., in the root module, or some other module imported by the root module).

```
@@filename(authors/authors.module)
@Module({
  imports: [PostsModule],
  providers: [AuthorsService, AuthorsResolver],
})
export class AuthorsModule {}
```

> info **Hint** It is helpful to organize your code by your so-called **domain model** (similar to the way you would organize entry points in a REST API). In this approach, keep your models (`ObjectType` classes), resolvers and services together within a Nest module representing the domain model. Keep all of these components in a single folder per module. When you do this, and use the Nest CLI to generate each element, Nest will wire all of these parts together (locating files in appropriate folders, generating entries in `provider` and `imports` arrays, etc.) automatically for you.

Mutations

Most discussions of GraphQL focus on data fetching, but any complete data platform needs a way to modify server-side data as well. In REST, any request could end up causing side-effects on the server, but best practice suggests we should not modify data in GET requests. GraphQL is similar - technically any query could be implemented to cause a data write. However, like REST, it's recommended to observe the convention that any operations that cause writes should be sent explicitly via a mutation (read more here).

The official Apollo documentation uses an `upvotePost()` mutation example. This mutation implements a method to increase a post's `votes` property value. To create an equivalent mutation in Nest, we'll make use of the `@Mutation()` decorator.

Code first

Let's add another method to the `AuthorResolver` used in the previous section (see resolvers).

```
@Mutation(returns => Post)
async upvotePost(@Args({ name: 'postId', type: () => Int }) postId:
number) {
  return this.postsService.upvoteById({ id: postId });
}
```

> info **Hint** All decorators (e.g., `@Resolver`, `@ResolveField`, `@Args`, etc.) are exported from the `@nestjs/graphql` package.

This will result in generating the following part of the GraphQL schema in SDL:

```
type Mutation {
  upvotePost(postId: Int!): Post
}
```

The `upvotePost()` method takes `postId` (`Int`) as an argument and returns an updated `Post` entity. For the reasons explained in the resolvers section, we have to explicitly set the expected type.

If the mutation needs to take an object as an argument, we can create an **input type**. The input type is a special kind of object type that can be passed in as an argument (read more here). To declare an input type, use the `@InputType()` decorator.

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

@InputType()
export class UpvotePostInput {
  @Field()
  postId: number;
}
```

> info **Hint** The `@InputType()` decorator takes an options object as an argument, so you can, for example, specify the input type's description. Note that, due to TypeScript's metadata reflection system limitations, you must either use the `@Field` decorator to manually indicate a type, or use a CLI plugin.

We can then use this type in the resolver class:

```
@Mutation(returns => Post)
async upvotePost(
  @Args('upvotePostData') upvotePostData: UpvotePostInput,
) {}
```

Schema first

Let's extend our `AuthorResolver` used in the previous section (see resolvers).

```
@Mutation()
async upvotePost(@Args('postId') postId: number) {
  return this.postsService.upvoteById({ id: postId });
}
```

Note that we assumed above that the business logic has been moved to the `PostsService` (querying the post and incrementing its `votes` property). The logic inside the `PostsService` class can be as simple or sophisticated as needed. The main point of this example is to show how resolvers can interact with other providers.

The last step is to add our mutation to the existing types definition.

```
type Author {
  id: Int!
  firstName: String
  lastName: String
  posts: [Post]
}

type Post {
  id: Int!
  title: String
  votes: Int
}

type Query {
  author(id: Int!): Author
}

type Mutation {
```

```
  upvotePost(postId: Int!): Post
}
```

The `upvotePost(postId: Int!): Post` mutation is now available to be called as part of our application's GraphQL API.

Subscriptions

In addition to fetching data using queries and modifying data using mutations, the GraphQL spec supports a third operation type, called `subscription`. GraphQL subscriptions are a way to push data from the server to the clients that choose to listen to real time messages from the server. Subscriptions are similar to queries in that they specify a set of fields to be delivered to the client, but instead of immediately returning a single answer, a channel is opened and a result is sent to the client every time a particular event happens on the server.

A common use case for subscriptions is notifying the client side about particular events, for example the creation of a new object, updated fields and so on (read more here).

Enable subscriptions with Apollo driver

To enable subscriptions, set the `installSubscriptionHandlers` property to `true`.

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  installSubscriptionHandlers: true,
}),
```

> warning **Warning** The `installSubscriptionHandlers` configuration option has been removed from the latest version of Apollo server and will be soon deprecated in this package as well. By default, `installSubscriptionHandlers` will fallback to use the `subscriptions-transport-ws` (read more) but we strongly recommend using the `graphql-ws`(read more) library instead.

To switch to use the `graphql-ws` package instead, use the following configuration:

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  subscriptions: {
    'graphql-ws': true
  },
}),
```

> info **Hint** You can also use both packages (`subscriptions-transport-ws` and `graphql-ws`) at the same time, for example, for backward compatibility.

Code first

To create a subscription using the code first approach, we use the `@Subscription()` decorator (exported from the `@nestjs/graphql` package) and the `PubSub` class from the `graphql-subscriptions` package, which provides a simple **publish/subscribe API**.

The following subscription handler takes care of **subscribing** to an event by calling `PubSub#asyncIterator`. This method takes a single argument, the `triggerName`, which corresponds to

an event topic name.

```
const pubSub = new PubSub();

@Resolver((of) => Author)
export class AuthorResolver {
  // ...
  @Subscription((returns) => Comment)
  commentAdded() {
    return pubSub.asyncIterator('commentAdded');
  }
}
```

> info **Hint** All decorators are exported from the `@nestjs/graphql` package, while the `PubSub` class is exported from the `graphql-subscriptions` package.

> warning **Note** `PubSub` is a class that exposes a simple `publish` and `subscribe API`. Read more about it here. Note that the Apollo docs warn that the default implementation is not suitable for production (read more here). Production apps should use a `PubSub` implementation backed by an external store (read more here).

This will result in generating the following part of the GraphQL schema in SDL:

```
type Subscription {
  commentAdded(): Comment!
}
```

Note that subscriptions, by definition, return an object with a single top level property whose key is the name of the subscription. This name is either inherited from the name of the subscription handler method (i.e., `commentAdded` above), or is provided explicitly by passing an option with the key `name` as the second argument to the `@Subscription()` decorator, as shown below.

```
@Subscription(returns => Comment, {
  name: 'commentAdded',
})
subscribeToCommentAdded() {
  return pubSub.asyncIterator('commentAdded');
}
```

This construct produces the same SDL as the previous code sample, but allows us to decouple the method name from the subscription.

Publishing

Now, to publish the event, we use the `PubSub#publish` method. This is often used within a mutation to trigger a client-side update when a part of the object graph has changed. For example:

```
@@filename(posts/posts.resolver)
@Mutation(returns => Post)
async addComment(
  @Args('postId', { type: () => Int }) postId: number,
  @Args('comment', { type: () => Comment }) comment: CommentInput,
) {
  const newComment = this.commentsService.addComment({ id: postId, comment
});
  pubSub.publish('commentAdded', { commentAdded: newComment });
  return newComment;
}
```

The `PubSub#publish` method takes a `triggerName` (again, think of this as an event topic name) as the first parameter, and an event payload as the second parameter. As mentioned, the subscription, by definition, returns a value and that value has a shape. Look again at the generated SDL for our `commentAdded` subscription:

```
type Subscription {
  commentAdded(): Comment!
}
```

This tells us that the subscription must return an object with a top-level property name of `commentAdded` that has a value which is a `Comment` object. The important point to note is that the shape of the event payload emitted by the `PubSub#publish` method must correspond to the shape of the value expected to return from the subscription. So, in our example above, the `pubSub.publish('commentAdded', {{ '{' }} commentAdded: newComment {{ '}' }})` statement publishes a `commentAdded` event with the appropriately shaped payload. If these shapes don't match, your subscription will fail during the GraphQL validation phase.

Filtering subscriptions

To filter out specific events, set the `filter` property to a filter function. This function acts similar to the function passed to an array `filter`. It takes two arguments: `payload` containing the event payload (as sent by the event publisher), and `variables` taking any arguments passed in during the subscription request. It returns a boolean determining whether this event should be published to client listeners.

```
@Subscription(returns => Comment, {
  filter: (payload, variables) =>
    payload.commentAdded.title === variables.title,
})
commentAdded(@Args('title') title: string) {
  return pubSub.asyncIterator('commentAdded');
}
```

Mutating subscription payloads

To mutate the published event payload, set the `resolve` property to a function. The function receives the event payload (as sent by the event publisher) and returns the appropriate value.

```
@Subscription(returns => Comment, {
  resolve: value => value,
})
commentAdded() {
  return pubSub.asyncIterator('commentAdded');
}
```

> warning **Note** If you use the `resolve` option, you should return the unwrapped payload (e.g., with our example, return a `newComment` object directly, not a `{{ '{' }} commentAdded: newComment {{ '}' }}` object).

If you need to access injected providers (e.g., use an external service to validate the data), use the following construction.

```
@Subscription(returns => Comment, {
  resolve(this: AuthorResolver, value) {
    // "this" refers to an instance of "AuthorResolver"
    return value;
  }
})
commentAdded() {
  return pubSub.asyncIterator('commentAdded');
}
```

The same construction works with filters:

```
@Subscription(returns => Comment, {
  filter(this: AuthorResolver, payload, variables) {
    // "this" refers to an instance of "AuthorResolver"
    return payload.commentAdded.title === variables.title;
  }
})
commentAdded() {
  return pubSub.asyncIterator('commentAdded');
}
```

Schema first

To create an equivalent subscription in Nest, we'll make use of the `@Subscription()` decorator.

```
const pubSub = new PubSub();
```

```
@Resolver('Author')
export class AuthorResolver {
  // ...
  @Subscription()
  commentAdded() {
    return pubSub.asyncIterator('commentAdded');
  }
}
```

To filter out specific events based on context and arguments, set the `filter` property.

```
@Subscription('commentAdded', {
  filter: (payload, variables) =>
    payload.commentAdded.title === variables.title,
})
commentAdded() {
  return pubSub.asyncIterator('commentAdded');
}
```

To mutate the published payload, we can use a `resolve` function.

```
@Subscription('commentAdded', {
  resolve: value => value,
})
commentAdded() {
  return pubSub.asyncIterator('commentAdded');
}
```

If you need to access injected providers (e.g., use an external service to validate the data), use the following construction:

```
@Subscription('commentAdded', {
  resolve(this: AuthorResolver, value) {
    // "this" refers to an instance of "AuthorResolver"
    return value;
  }
})
commentAdded() {
  return pubSub.asyncIterator('commentAdded');
}
```

The same construction works with filters:

```
@Subscription('commentAdded', {
  filter(this: AuthorResolver, payload, variables) {
```

```
      // "this" refers to an instance of "AuthorResolver"
      return payload.commentAdded.title === variables.title;
    }
  })
  commentAdded() {
    return pubSub.asyncIterator('commentAdded');
  }
```

The last step is to update the type definitions file.

```
type Author {
  id: Int!
  firstName: String
  lastName: String
  posts: [Post]
}

type Post {
  id: Int!
  title: String
  votes: Int
}

type Query {
  author(id: Int!): Author
}

type Comment {
  id: String
  content: String
}

type Subscription {
  commentAdded(title: String!): Comment
}
```

With this, we've created a single `commentAdded(title: String!): Comment` subscription. You can find a full sample implementation here.

PubSub

We instantiated a local `PubSub` instance above. The preferred approach is to define `PubSub` as a provider and inject it through the constructor (using the `@Inject()` decorator). This allows us to re-use the instance across the whole application. For example, define a provider as follows, then inject `'PUB_SUB'` where needed.

```
{
  provide: 'PUB_SUB',
```

```
  useValue: new PubSub(),
}
```

Customize subscriptions server

To customize the subscriptions server (e.g., change the path), use the `subscriptions` options property.

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  subscriptions: {
    'subscriptions-transport-ws': {
      path: '/graphql'
    },
  }
}),
```

If you're using the `graphql-ws` package for subscriptions, replace the `subscriptions-transport-ws` key with `graphql-ws`, as follows:

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  subscriptions: {
    'graphql-ws': {
      path: '/graphql'
    },
  }
}),
```

Authentication over WebSockets

Checking whether the user is authenticated can be done inside the `onConnect` callback function that you can specify in the `subscriptions` options.

The `onConnect` will receive as a first argument the `connectionParams` passed to the `SubscriptionClient` (read more).

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  subscriptions: {
    'subscriptions-transport-ws': {
      onConnect: (connectionParams) => {
        const authToken = connectionParams.authToken;
        if (!isValid(authToken)) {
          throw new Error('Token is not valid');
        }
        // extract user information from token
```

```
        const user = parseToken(authToken);
        // return user info to add them to the context later
        return { user };
      },
    }
  },
  context: ({ connection }) => {
    // connection.context will be equal to what was returned by the
"onConnect" callback
  },
}),
```

The `authToken` in this example is only sent once by the client, when the connection is first established. All subscriptions made with this connection will have the same `authToken`, and thus the same user info.

> warning **Note** There is a bug in `subscriptions-transport-ws` that allows connections to skip the `onConnect` phase (read more). You should not assume that `onConnect` was called when the user starts a subscription, and always check that the `context` is populated.

If you're using the `graphql-ws` package, the signature of the `onConnect` callback will be slightly different:

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  subscriptions: {
    'graphql-ws': {
      onConnect: (context: Context<any>) => {
        const { connectionParams, extra } = context;
        // user validation will remain the same as in the example above
        // when using with graphql-ws, additional context value should be
stored in the extra field
        extra.user = { user: {} };
      },
    },
  },
  context: ({ extra }) => {
    // you can now access your additional context value through the extra
field
  },
});
```

Enable subscriptions with Mercurius driver

To enable subscriptions, set the `subscription` property to `true`.

```
GraphQLModule.forRoot<MercuriusDriverConfig>({
  driver: MercuriusDriver,
  subscription: true,
}),
```

> info **Hint** You can also pass the options object to set up a custom emitter, validate incoming connections, etc. Read more here (see `subscription`).

Code first

To create a subscription using the code first approach, we use the `@Subscription()` decorator (exported from the `@nestjs/graphql` package) and the `PubSub` class from the `mercurius` package, which provides a simple **publish/subscribe API**.

The following subscription handler takes care of **subscribing** to an event by calling `PubSub#asyncIterator`. This method takes a single argument, the `triggerName`, which corresponds to an event topic name.

```
@Resolver((of) => Author)
export class AuthorResolver {
  // ...
  @Subscription((returns) => Comment)
  commentAdded(@Context('pubsub') pubSub: PubSub) {
    return pubSub.subscribe('commentAdded');
  }
}
```

> info **Hint** All decorators used in the example above are exported from the `@nestjs/graphql` package, while the `PubSub` class is exported from the `mercurius` package.

> warning **Note** `PubSub` is a class that exposes a simple `publish` and `subscribe` API. Check out this section on how to register a custom `PubSub` class.

This will result in generating the following part of the GraphQL schema in SDL:

```
type Subscription {
  commentAdded(): Comment!
}
```

Note that subscriptions, by definition, return an object with a single top level property whose key is the name of the subscription. This name is either inherited from the name of the subscription handler method (i.e., `commentAdded` above), or is provided explicitly by passing an option with the key `name` as the second argument to the `@Subscription()` decorator, as shown below.

```
@Subscription(returns => Comment, {
  name: 'commentAdded',
})
subscribeToCommentAdded(@Context('pubsub') pubSub: PubSub) {
  return pubSub.subscribe('commentAdded');
}
```

This construct produces the same SDL as the previous code sample, but allows us to decouple the method name from the subscription.

Publishing

Now, to publish the event, we use the `PubSub#publish` method. This is often used within a mutation to trigger a client-side update when a part of the object graph has changed. For example:

```
@@filename(posts/posts.resolver)
@Mutation(returns => Post)
async addComment(
  @Args('postId', { type: () => Int }) postId: number,
  @Args('comment', { type: () => Comment }) comment: CommentInput,
  @Context('pubsub') pubSub: PubSub,
) {
  const newComment = this.commentsService.addComment({ id: postId, comment
});
  await pubSub.publish({
    topic: 'commentAdded',
    payload: {
      commentAdded: newComment
    }
  });
  return newComment;
}
```

As mentioned, the subscription, by definition, returns a value and that value has a shape. Look again at the generated SDL for our `commentAdded` subscription:

```
type Subscription {
  commentAdded(): Comment!
}
```

This tells us that the subscription must return an object with a top-level property name of `commentAdded` that has a value which is a `Comment` object. The important point to note is that the shape of the event payload emitted by the `PubSub#publish` method must correspond to the shape of the value expected to return from the subscription. So, in our example above, the `pubSub.publish({{ '{' }} topic: 'commentAdded', payload: {{ '{' }} commentAdded: newComment {{ '}' }} {{ '}' }})` statement publishes a `commentAdded` event with the appropriately shaped payload. If these shapes don't match, your subscription will fail during the GraphQL validation phase.

Filtering subscriptions

To filter out specific events, set the `filter` property to a filter function. This function acts similar to the function passed to an array `filter`. It takes two arguments: `payload` containing the event payload (as sent by the event publisher), and `variables` taking any arguments passed in during the subscription request. It returns a boolean determining whether this event should be published to client listeners.

```
@Subscription(returns => Comment, {
  filter: (payload, variables) =>
    payload.commentAdded.title === variables.title,
})
commentAdded(@Args('title') title: string, @Context('pubsub') pubSub:
PubSub) {
  return pubSub.subscribe('commentAdded');
}
```

If you need to access injected providers (e.g., use an external service to validate the data), use the following construction.

```
@Subscription(returns => Comment, {
  filter(this: AuthorResolver, payload, variables) {
    // "this" refers to an instance of "AuthorResolver"
    return payload.commentAdded.title === variables.title;
  }
})
commentAdded(@Args('title') title: string, @Context('pubsub') pubSub:
PubSub) {
  return pubSub.subscribe('commentAdded');
}
```

Schema first

To create an equivalent subscription in Nest, we'll make use of the `@Subscription()` decorator.

```
const pubSub = new PubSub();

@Resolver('Author')
export class AuthorResolver {
  // ...
  @Subscription()
  commentAdded(@Context('pubsub') pubSub: PubSub) {
    return pubSub.subscribe('commentAdded');
  }
}
```

To filter out specific events based on context and arguments, set the `filter` property.

```
@Subscription('commentAdded', {
  filter: (payload, variables) =>
    payload.commentAdded.title === variables.title,
})
commentAdded(@Context('pubsub') pubSub: PubSub) {
```

```
    return pubSub.subscribe('commentAdded');
  }
```

If you need to access injected providers (e.g., use an external service to validate the data), use the following construction:

```
@Subscription('commentAdded', {
  filter(this: AuthorResolver, payload, variables) {
    // "this" refers to an instance of "AuthorResolver"
    return payload.commentAdded.title === variables.title;
  }
})
commentAdded(@Context('pubsub') pubSub: PubSub) {
  return pubSub.subscribe('commentAdded');
}
```

The last step is to update the type definitions file.

```
type Author {
  id: Int!
  firstName: String
  lastName: String
  posts: [Post]
}

type Post {
  id: Int!
  title: String
  votes: Int
}

type Query {
  author(id: Int!): Author
}

type Comment {
  id: String
  content: String
}

type Subscription {
  commentAdded(title: String!): Comment
}
```

With this, we've created a single `commentAdded(title: String!): Comment` subscription.

PubSub

In the examples above, we used the default `PubSub` emitter (mqemitter) The preferred approach (for production) is to use `mqemitter-redis`. Alternatively, a custom `PubSub` implementation can be provided (read more here)

```
GraphQLModule.forRoot<MercuriusDriverConfig>({
  driver: MercuriusDriver,
  subscription: {
    emitter: require('mqemitter-redis')({
      port: 6579,
      host: '127.0.0.1',
    }),
  },
});
```

Authentication over WebSockets

Checking whether the user is authenticated can be done inside the `verifyClient` callback function that you can specify in the `subscription` options.

The `verifyClient` will receive the `info` object as a first argument which you can use to retrieve the request's headers.

```
GraphQLModule.forRoot<MercuriusDriverConfig>({
  driver: MercuriusDriver,
  subscription: {
    verifyClient: (info, next) => {
      const authorization = info.req.headers?.authorization as string;
      if (!authorization?.startsWith('Bearer ')) {
        return next(false);
      }
      next(true);
    },
  }
}),
```

Scalars

A GraphQL object type has a name and fields, but at some point those fields have to resolve to some concrete data. That's where the scalar types come in: they represent the leaves of the query (read more here). GraphQL includes the following default types: `Int`, `Float`, `String`, `Boolean` and `ID`. In addition to these built-in types, you may need to support custom atomic data types (e.g., `Date`).

Code first

The code-first approach ships with five scalars in which three of them are simple aliases for the existing GraphQL types.

- `ID` (alias for `GraphQLID`) - represents a unique identifier, often used to refetch an object or as the key for a cache
- `Int` (alias for `GraphQLInt`) - a signed 32-bit integer
- `Float` (alias for `GraphQLFloat`) - a signed double-precision floating-point value
- `GraphQLISODateTime` - a date-time string at UTC (used by default to represent `Date` type)
- `GraphQLTimestamp` - a signed integer which represents date and time as number of milliseconds from start of UNIX epoch

The `GraphQLISODateTime` (e.g. `2019-12-03T09:54:33Z`) is used by default to represent the `Date` type. To use the `GraphQLTimestamp` instead, set the `dateScalarMode` of the `buildSchemaOptions` object to `'timestamp'` as follows:

```
GraphQLModule.forRoot({
  buildSchemaOptions: {
    dateScalarMode: 'timestamp',
  }
}),
```

Likewise, the `GraphQLFloat` is used by default to represent the `number` type. To use the `GraphQLInt` instead, set the `numberScalarMode` of the `buildSchemaOptions` object to `'integer'` as follows:

```
GraphQLModule.forRoot({
  buildSchemaOptions: {
    numberScalarMode: 'integer',
  }
}),
```

In addition, you can create custom scalars.

Override a default scalar

To create a custom implementation for the `Date` scalar, simply create a new class.

```
import { Scalar, CustomScalar } from '@nestjs/graphql';
import { Kind, ValueNode } from 'graphql';

@Scalar('Date', (type) => Date)
export class DateScalar implements CustomScalar<number, Date> {
  description = 'Date custom scalar type';

  parseValue(value: number): Date {
    return new Date(value); // value from the client
  }

  serialize(value: Date): number {
    return value.getTime(); // value sent to the client
  }

  parseLiteral(ast: ValueNode): Date {
    if (ast.kind === Kind.INT) {
      return new Date(ast.value);
    }
    return null;
  }
}
```

With this in place, register `DateScalar` as a provider.

```
@Module({
  providers: [DateScalar],
})
export class CommonModule {}
```

Now we can use the `Date` type in our classes.

```
@Field()
creationDate: Date;
```

Import a custom scalar

To use a custom scalar, import and register it as a resolver. We'll use the `graphql-type-json` package for demonstration purposes. This npm package defines a `JSON` GraphQL scalar type.

Start by installing the package:

```
$ npm i --save graphql-type-json
```

Once the package is installed, we pass a custom resolver to the `forRoot()` method:

```
import GraphQLJSON from 'graphql-type-json';

@Module({
  imports: [
    GraphQLModule.forRoot({
      resolvers: { JSON: GraphQLJSON },
    }),
  ],
})
export class AppModule {}
```

Now we can use the `JSON` type in our classes.

```
@Field((type) => GraphQLJSON)
info: JSON;
```

For a suite of useful scalars, take a look at the graphql-scalars package.

Create a custom scalar

To define a custom scalar, create a new `GraphQLScalarType` instance. We'll create a custom `UUID` scalar.

```
const regex = /^[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]
{12}$/i;

function validate(uuid: unknown): string | never {
  if (typeof uuid !== "string" || !regex.test(uuid)) {
    throw new Error("invalid uuid");
  }
  return uuid;
}

export const CustomUuidScalar = new GraphQLScalarType({
  name: 'UUID',
  description: 'A simple UUID parser',
  serialize: (value) => validate(value),
  parseValue: (value) => validate(value),
  parseLiteral: (ast) => validate(ast.value)
})
```

We pass a custom resolver to the `forRoot()` method:

```
@Module({
  imports: [
    GraphQLModule.forRoot({
      resolvers: { UUID: CustomUuidScalar },
```

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

Now we can use the `UUID` type in our classes.

```
@Field((type) => CustomUuidScalar)
uuid: string;
```

Schema first

To define a custom scalar (read more about scalars here), create a type definition and a dedicated resolver. Here (as in the official documentation), we'll use the `graphql-type-json` package for demonstration purposes. This npm package defines a `JSON` GraphQL scalar type.

Start by installing the package:

```
$ npm i --save graphql-type-json
```

Once the package is installed, we pass a custom resolver to the `forRoot()` method:

```
import GraphQLJSON from 'graphql-type-json';

@Module({
  imports: [
    GraphQLModule.forRoot({
      typePaths: ['./**/*.graphql'],
      resolvers: { JSON: GraphQLJSON },
    }),
  ],
})
export class AppModule {}
```

Now we can use the `JSON` scalar in our type definitions:

```
scalar JSON

type Foo {
  field: JSON
}
```

Another method to define a scalar type is to create a simple class. Assume we want to enhance our schema with the `Date` type.

```
import { Scalar, CustomScalar } from '@nestjs/graphql';
import { Kind, ValueNode } from 'graphql';

@Scalar('Date')
export class DateScalar implements CustomScalar<number, Date> {
  description = 'Date custom scalar type';

  parseValue(value: number): Date {
    return new Date(value); // value from the client
  }

  serialize(value: Date): number {
    return value.getTime(); // value sent to the client
  }

  parseLiteral(ast: ValueNode): Date {
    if (ast.kind === Kind.INT) {
      return new Date(ast.value);
    }
    return null;
  }
}
```

With this in place, register `DateScalar` as a provider.

```
@Module({
  providers: [DateScalar],
})
export class CommonModule {}
```

Now we can use the `Date` scalar in type definitions.

```
scalar Date
```

By default, the generated TypeScript definition for all scalars is `any` - which isn't particularly typesafe. But, you can configure how Nest generates typings for your custom scalars when you specify how to generate types:

```
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',
  defaultScalarType: 'unknown',
  customScalarTypeMapping: {
    DateTime: 'Date',
    BigNumber: '_BigNumber',
  },
  additionalHeader: "import _BigNumber from 'bignumber.js'",
});
```

> info **Hint** Alternatively, you can use a type reference instead, for example: `DateTime: Date`. In this case, `GraphQLDefinitionsFactory` will extract the name property of the specified type (`Date.name`) to generate TS definitions. Note: adding an import statement for non-built-in types (custom types) is required.

Now, given the following GraphQL custom scalar types:

```
scalar DateTime
scalar BigNumber
scalar Payload
```

We will now see the following generated TypeScript definitions in `src/graphql.ts`:

```
import _BigNumber from 'bignumber.js';

export type DateTime = Date;
export type BigNumber = _BigNumber;
export type Payload = unknown;
```

Here, we've used the `customScalarTypeMapping` property to supply a map of the types we wish to declare for our custom scalars. We've also provided an `additionalHeader` property so that we can add any imports required for these type definitions. Lastly, we've added a `defaultScalarType` of `'unknown'`, so that any custom scalars not specified in `customScalarTypeMapping` will be aliased to `unknown` instead of `any` (which TypeScript recommends using since 3.0 for added type safety).

> info **Hint** Note that we've imported `_BigNumber` from `bignumber.js`; this is to avoid circular type references.

Directives

A directive can be attached to a field or fragment inclusion, and can affect execution of the query in any way the server desires (read more here). The GraphQL specification provides several default directives:

- `@include(if: Boolean)` - only include this field in the result if the argument is true
- `@skip(if: Boolean)` - skip this field if the argument is true
- `@deprecated(reason: String)` - marks field as deprecated with message

A directive is an identifier preceded by a `@` character, optionally followed by a list of named arguments, which can appear after almost any element in the GraphQL query and schema languages.

Custom directives

To instruct what should happen when Apollo/Mercurius encounters your directive, you can create a transformer function. This function uses the `mapSchema` function to iterate through locations in your schema (field definitions, type definitions, etc.) and perform corresponding transformations.

```
import { getDirective, MapperKind, mapSchema } from '@graphql-
tools/utils';
import { defaultFieldResolver, GraphQLSchema } from 'graphql';

export function upperDirectiveTransformer(
  schema: GraphQLSchema,
  directiveName: string,
) {
  return mapSchema(schema, {
    [MapperKind.OBJECT_FIELD]: (fieldConfig) => {
      const upperDirective = getDirective(
        schema,
        fieldConfig,
        directiveName,
      )?.[0];

      if (upperDirective) {
        const { resolve = defaultFieldResolver } = fieldConfig;

        // Replace the original resolver with a function that *first*
calls
        // the original resolver, then converts its result to upper case
        fieldConfig.resolve = async function (source, args, context, info)
{
          const result = await resolve(source, args, context, info);
          if (typeof result === 'string') {
            return result.toUpperCase();
          }
          return result;
        };
        return fieldConfig;
      }
    },
```

```
  });
}
```

Now, apply the `upperDirectiveTransformer` transformation function in the `GraphQLModule#forRoot` method using the `transformSchema` function:

```
GraphQLModule.forRoot({
  // ...
  transformSchema: (schema) => upperDirectiveTransformer(schema, 'upper'),
});
```

Once registered, the `@upper` directive can be used in our schema. However, the way you apply the directive will vary depending on the approach you use (code first or schema first).

Code first

In the code first approach, use the `@Directive()` decorator to apply the directive.

```
@Directive('@upper')
@Field()
title: string;
```

> info **Hint** The `@Directive()` decorator is exported from the `@nestjs/graphql` package.

Directives can be applied on fields, field resolvers, input and object types, as well as queries, mutations, and subscriptions. Here's an example of the directive applied on the query handler level:

```
@Directive('@deprecated(reason: "This query will be removed in the next
version")')
@Query(returns => Author, { name: 'author' })
async getAuthor(@Args({ name: 'id', type: () => Int }) id: number) {
  return this.authorsService.findOneById(id);
}
```

> warn **Warning** Directives applied through the `@Directive()` decorator will not be reflected in the generated schema definition file.

Lastly, make sure to declare directives in the `GraphQLModule`, as follows:

```
GraphQLModule.forRoot({
  // ...,
  transformSchema: schema => upperDirectiveTransformer(schema, 'upper'),
  buildSchemaOptions: {
    directives: [
```

```
        new GraphQLDirective({
          name: 'upper',
          locations: [DirectiveLocation.FIELD_DEFINITION],
        }),
      ],
    },
  }),
```

> info **Hint** Both `GraphQLDirective` and `DirectiveLocation` are exported from the `graphql` package.

Schema first

In the schema first approach, apply directives directly in SDL.

```
directive @upper on FIELD_DEFINITION

type Post {
  id: Int!
  title: String! @upper
  votes: Int
}
```

Interfaces

Like many type systems, GraphQL supports interfaces. An **Interface** is an abstract type that includes a certain set of fields that a type must include to implement the interface (read more here).

Code first

When using the code first approach, you define a GraphQL interface by creating an abstract class annotated with the `@InterfaceType()` decorator exported from the `@nestjs/graphql`.

```
import { Field, ID, InterfaceType } from '@nestjs/graphql';

@InterfaceType()
export abstract class Character {
  @Field((type) => ID)
  id: string;

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

> warning **Warning** TypeScript interfaces cannot be used to define GraphQL interfaces.

This will result in generating the following part of the GraphQL schema in SDL:

```
interface Character {
  id: ID!
  name: String!
}
```

Now, to implement the `Character` interface, use the `implements` key:

```
@ObjectType({
  implements: () => [Character],
})
export class Human implements Character {
  id: string;
  name: string;
}
```

> info **Hint** The `@ObjectType()` decorator is exported from the `@nestjs/graphql` package.

The default `resolveType()` function generated by the library extracts the type based on the value returned from the resolver method. This means that you must return class instances (you cannot return literal JavaScript objects).

To provide a customized `resolveType()` function, pass the `resolveType` property to the options object passed into the `@InterfaceType()` decorator, as follows:

```
@InterfaceType({
  resolveType(book) {
    if (book.colors) {
      return ColoringBook;
    }
    return TextBook;
  },
})
export abstract class Book {
  @Field((type) => ID)
  id: string;

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

Interface resolvers

So far, using interfaces, you could only share field definitions with your objects. If you also want to share the actual field resolvers implementation, you can create a dedicated interface resolver, as follows:

```
import { Resolver, ResolveField, Parent, Info } from '@nestjs/graphql';

@Resolver(type => Character) // Reminder: Character is an interface
export class CharacterInterfaceResolver {
  @ResolveField(() => [Character])
  friends(
    @Parent() character, // Resolved object that implements Character
    @Info() { parentType }, // Type of the object that implements
Character
    @Args('search', { type: () => String }) searchTerm: string,
  ) {
    // Get character's friends
    return [];
  }
}
```

Now the `friends` field resolver is auto-registered for all object types that implement the `Character` interface.

Schema first

To define an interface in the schema first approach, simply create a GraphQL interface with SDL.

```
interface Character {
  id: ID!
  name: String!
}
```

Then, you can use the typings generation feature (as shown in the quick start chapter) to generate corresponding TypeScript definitions:

```
export interface Character {
  id: string;
  name: string;
}
```

Interfaces require an extra `__resolveType` field in the resolver map to determine which type the interface should resolve to. Let's create a `CharactersResolver` class and define the `__resolveType` method:

```
@Resolver('Character')
export class CharactersResolver {
  @ResolveField()
  __resolveType(value) {
    if ('age' in value) {
      return Person;
    }
    return null;
  }
}
```

> info **Hint** All decorators are exported from the `@nestjs/graphql` package.

Unions

Union types are very similar to interfaces, but they don't get to specify any common fields between the types (read more here). Unions are useful for returning disjoint data types from a single field.

Code first

To define a GraphQL union type, we must define classes that this union will be composed of. Following the example from the Apollo documentation, we'll create two classes. First, `Book`:

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

@ObjectType()
export class Book {
  @Field()
  title: string;
}
```

And then `Author`:

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

@ObjectType()
export class Author {
  @Field()
  name: string;
}
```

With this in place, register the `ResultUnion` union using the `createUnionType` function exported from the `@nestjs/graphql` package:

```
export const ResultUnion = createUnionType({
  name: 'ResultUnion',
  types: () => [Author, Book] as const,
});
```

> warning **Warning** The array returned by the `types` property of the `createUnionType` function should be given a const assertion. If the const assertion is not given, a wrong declaration file will be generated at compile time, and an error will occur when using it from another project.

Now, we can reference the `ResultUnion` in our query:

```
@Query(returns => [ResultUnion])
search(): Array<typeof ResultUnion> {
```

```
    return [new Author(), new Book()];
  }
```

This will result in generating the following part of the GraphQL schema in SDL:

```
type Author {
  name: String!
}

type Book {
  title: String!
}

union ResultUnion = Author | Book

type Query {
  search: [ResultUnion!]!
}
```

The default `resolveType()` function generated by the library will extract the type based on the value returned from the resolver method. That means returning class instances instead of literal JavaScript object is obligatory.

To provide a customized `resolveType()` function, pass the `resolveType` property to the options object passed into the `createUnionType()` function, as follows:

```
export const ResultUnion = createUnionType({
  name: 'ResultUnion',
  types: () => [Author, Book] as const,
  resolveType(value) {
    if (value.name) {
      return Author;
    }
    if (value.title) {
      return Book;
    }
    return null;
  },
});
```

Schema first

To define a union in the schema first approach, simply create a GraphQL union with SDL.

```
type Author {
  name: String!
```

```
}

type Book {
  title: String!
}

union ResultUnion = Author | Book
```

Then, you can use the typings generation feature (as shown in the quick start chapter) to generate corresponding TypeScript definitions:

```
export class Author {
  name: string;
}

export class Book {
  title: string;
}

export type ResultUnion = Author | Book;
```

Unions require an extra `__resolveType` field in the resolver map to determine which type the union should resolve to. Also, note that the `ResultUnionResolver` class has to be registered as a provider in any module. Let's create a `ResultUnionResolver` class and define the `__resolveType` method.

```
@Resolver('ResultUnion')
export class ResultUnionResolver {
  @ResolveField()
  __resolveType(value) {
    if (value.name) {
      return 'Author';
    }
    if (value.title) {
      return 'Book';
    }
    return null;
  }
}
```

> info **Hint** All decorators are exported from the `@nestjs/graphql` package.

Enums

Enumeration types are a special kind of scalar that is restricted to a particular set of allowed values (read more here). This allows you to:

- validate that any arguments of this type are one of the allowed values

- communicate through the type system that a field will always be one of a finite set of values

Code first

When using the code first approach, you define a GraphQL enum type by simply creating a TypeScript enum.

```
export enum AllowedColor {
  RED,
  GREEN,
  BLUE,
}
```

With this in place, register the `AllowedColor` enum using the `registerEnumType` function exported from the `@nestjs/graphql` package:

```
registerEnumType(AllowedColor, {
  name: 'AllowedColor',
});
```

Now you can reference the `AllowedColor` in our types:

```
@Field(type => AllowedColor)
favoriteColor: AllowedColor;
```

This will result in generating the following part of the GraphQL schema in SDL:

```
enum AllowedColor {
  RED
  GREEN
  BLUE
}
```

To provide a description for the enum, pass the `description` property into the `registerEnumType()` function.

```
registerEnumType(AllowedColor, {
  name: 'AllowedColor',
  description: 'The supported colors.',
});
```

To provide a description for the enum values, or to mark a value as deprecated, pass the `valuesMap` property, as follows:

```
registerEnumType(AllowedColor, {
  name: 'AllowedColor',
  description: 'The supported colors.',
  valuesMap: {
    RED: {
      description: 'The default color.',
    },
    BLUE: {
      deprecationReason: 'Too blue.',
    },
  },
});
```

This will generate the following GraphQL schema in SDL:

```
"""
The supported colors.
"""
enum AllowedColor {
  """
  The default color.
  """
  RED
  GREEN
  BLUE @deprecated(reason: "Too blue.")
}
```

Schema first

To define an enumerator in the schema first approach, simply create a GraphQL enum with SDL.

```
enum AllowedColor {
  RED
  GREEN
  BLUE
}
```

Then you can use the typings generation feature (as shown in the quick start chapter) to generate corresponding TypeScript definitions:

```
export enum AllowedColor {
  RED
```

```
  GREEN
  BLUE
}
```

Sometimes a backend forces a different value for an enum internally than in the public API. In this example the API contains `RED`, however in resolvers we may use `#f00` instead (read more here). To accomplish this, declare a resolver object for the `AllowedColor` enum:

```
export const allowedColorResolver: Record<keyof typeof AllowedColor, any>
= {
  RED: '#f00',
};
```

> info **Hint** All decorators are exported from the `@nestjs/graphql` package.

Then use this resolver object together with the `resolvers` property of the `GraphQLModule#forRoot()` method, as follows:

```
GraphQLModule.forRoot({
  resolvers: {
    AllowedColor: allowedColorResolver,
  },
});
```

Field middleware

> warning **Warning** This chapter applies only to the code first approach.

Field Middleware lets you run arbitrary code **before or after** a field is resolved. A field middleware can be used to convert the result of a field, validate the arguments of a field, or even check field-level roles (for example, required to access a target field for which a middleware function is executed).

You can connect multiple middleware functions to a field. In this case, they will be called sequentially along the chain where the previous middleware decides to call the next one. The order of the middleware functions in the `middleware` array is important. The first resolver is the "most-outer" layer, so it gets executed first and last (similarly to the `graphql-middleware` package). The second resolver is the "second-outer" layer, so it gets executed second and second to last.

Getting started

Let's start off by creating a simple middleware that will log a field value before it's sent back to the client:

```
import { FieldMiddleware, MiddlewareContext, NextFn } from
'@nestjs/graphql';

const loggerMiddleware: FieldMiddleware = async (
  ctx: MiddlewareContext,
  next: NextFn,
) => {
  const value = await next();
  console.log(value);
  return value;
};
```

> info **Hint** The `MiddlewareContext` is an object that consist of the same arguments that are normally received by the GraphQL resolver function (`{{ '{' }} source, args, context, info {{ '}' }}`), while `NextFn` is a function that let you execute the next middleware in the stack (bound to this field) or the actual field resolver.

> warning **Warning** Field middleware functions cannot inject dependencies nor access Nest's DI container as they are designed to be very lightweight and shouldn't perform any potentially time-consuming operations (like retrieving data from the database). If you need to call external services/query data from the data source, you should do it in a guard/interceptor bounded to a root query/mutation handler and assign it to `context` object which you can access from within the field middleware (specifically, from the `MiddlewareContext` object).

Note that field middleware must match the `FieldMiddleware` interface. In the example above, we first run the `next()` function (which executes the actual field resolver and returns a field value) and then, we log this value to our terminal. Also, the value returned from the middleware function completely overrides the previous value and since we don't want to perform any changes, we simply return the original value.

With this in place, we can register our middleware directly in the `@Field()` decorator, as follows:

```
@ObjectType()
export class Recipe {
  @Field({ middleware: [loggerMiddleware] })
  title: string;
}
```

Now whenever we request the `title` field of `Recipe` object type, the original field's value will be logged to the console.

> info **Hint** To learn how you can implement a field-level permissions system with the use of extensions feature, check out this section.

> warning **Warning** Field middleware can be applied only to `ObjectType` classes. For more details, check out this issue.

Also, as mentioned above, we can control the field's value from within the middleware function. For demonstration purposes, let's capitalise a recipe's title (if present):

```
const value = await next();
return value?.toUpperCase();
```

In this case, every title will be automatically uppercased, when requested.

Likewise, you can bind a field middleware to a custom field resolver (a method annotated with the `@ResolveField()` decorator), as follows:

```
@ResolveField(() => String, { middleware: [loggerMiddleware] })
title() {
  return 'Placeholder';
}
```

> warning **Warning** In case enhancers are enabled at the field resolver level (read more), field middleware functions will run before any interceptors, guards, etc., **bounded to the method** (but after the root-level enhancers registered for query or mutation handlers).

Global field middleware

In addition to binding a middleware directly to a specific field, you can also register one or multiple middleware functions globally. In this case, they will be automatically connected to all fields of your object types.

```
GraphQLModule.forRoot({
  autoSchemaFile: 'schema.gql',
  buildSchemaOptions: {
    fieldMiddleware: [loggerMiddleware],
```

```
    },
  }),
```

> info **Hint** Globally registered field middleware functions will be executed **before** locally registered ones (those bound directly to specific fields).

Mapped types

> warning **Warning** This chapter applies only to the code first approach.

As you build out features like CRUD (Create/Read/Update/Delete) it's often useful to construct variants on a base entity type. Nest provides several utility functions that perform type transformations to make this task more convenient.

Partial

When building input validation types (also called Data Transfer Objects or DTOs), it's often useful to build **create** and **update** variations on the same type. For example, the **create** variant may require all fields, while the **update** variant may make all fields optional.

Nest provides the `PartialType()` utility function to make this task easier and minimize boilerplate.

The `PartialType()` function returns a type (class) with all the properties of the input type set to optional. For example, suppose we have a **create** type as follows:

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

  @Field()
  password: string;

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

By default, all of these fields are required. To create a type with the same fields, but with each one optional, use `PartialType()` passing the class reference (`CreateUserInput`) as an argument:

```
@InputType()
export class UpdateUserInput extends PartialType(CreateUserInput) {}
```

> info **Hint** The `PartialType()` function is imported from the `@nestjs/graphql` package.

The `PartialType()` function takes an optional second argument that is a reference to a decorator factory. This argument can be used to change the decorator function applied to the resulting (child) class. If not specified, the child class effectively uses the same decorator as the **parent** class (the class referenced in the first argument). In the example above, we are extending `CreateUserInput` which is annotated with the `@InputType()` decorator. Since we want `UpdateUserInput` to also be treated as if it were decorated with `@InputType()`, we didn't need to pass `InputType` as the second argument. If the parent and child

types are different, (e.g., the parent is decorated with `@ObjectType`), we would pass `InputType` as the second argument. For example:

```
@InputType()
export class UpdateUserInput extends PartialType(User, InputType) {}
```

Pick

The `PickType()` function constructs a new type (class) by picking a set of properties from an input type. For example, suppose we start with a type like:

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

  @Field()
  password: string;

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

We can pick a set of properties from this class using the `PickType()` utility function:

```
@InputType()
export class UpdateEmailInput extends PickType(CreateUserInput, [
  'email',
] as const) {}
```

> info **Hint** The `PickType()` function is imported from the `@nestjs/graphql` package.

Omit

The `OmitType()` function constructs a type by picking all properties from an input type and then removing a particular set of keys. For example, suppose we start with a type like:

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

  @Field()
  password: string;
```

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

We can generate a derived type that has every property **except** `email` as shown below. In this construct, the second argument to `OmitType` is an array of property names.

```
@InputType()
export class UpdateUserInput extends OmitType(CreateUserInput, [
  'email',
] as const) {}
```

> info **Hint** The `OmitType()` function is imported from the `@nestjs/graphql` package.

Intersection

The `IntersectionType()` function combines two types into one new type (class). For example, suppose we start with two types like:

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

  @Field()
  password: string;
}

@ObjectType()
export class AdditionalUserInfo {
  @Field()
  firstName: string;

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

We can generate a new type that combines all properties in both types.

```
@InputType()
export class UpdateUserInput extends IntersectionType(
  CreateUserInput,
  AdditionalUserInfo,
) {}
```

> info **Hint** The `IntersectionType()` function is imported from the `@nestjs/graphql` package.

Composition

The type mapping utility functions are composable. For example, the following will produce a type (class) that has all of the properties of the `CreateUserInput` type except for `email`, and those properties will be set to optional:

```
@InputType()
export class UpdateUserInput extends PartialType(
  OmitType(CreateUserInput, ['email'] as const),
) {}
```

Plugins with Apollo

Plugins enable you to extend Apollo Server's core functionality by performing custom operations in response to certain events. Currently, these events correspond to individual phases of the GraphQL request lifecycle, and to the startup of Apollo Server itself (read more here). For example, a basic logging plugin might log the GraphQL query string associated with each request that's sent to Apollo Server.

Custom plugins

To create a plugin, declare a class annotated with the `@Plugin` decorator exported from the `@nestjs/apollo` package. Also, for better code autocompletion, implement the `ApolloServerPlugin` interface from the `@apollo/server` package.

```
import { ApolloServerPlugin, GraphQLRequestListener } from
'@apollo/server';
import { Plugin } from '@nestjs/apollo';

@Plugin()
export class LoggingPlugin implements ApolloServerPlugin {
  async requestDidStart(): Promise<GraphQLRequestListener<any>> {
    console.log('Request started');
    return {
      async willSendResponse() {
        console.log('Will send response');
      },
    };
  }
}
```

With this in place, we can register the `LoggingPlugin` as a provider.

```
@Module({
  providers: [LoggingPlugin],
})
export class CommonModule {}
```

Nest will automatically instantiate a plugin and apply it to the Apollo Server.

Using external plugins

There are several plugins provided out-of-the-box. To use an existing plugin, simply import it and add it to the `plugins` array:

```
GraphQLModule.forRoot({
  // ...
```

```
    plugins: [ApolloServerOperationRegistry({ /* options */})]
  }),
```

> info **Hint** The `ApolloServerOperationRegistry` plugin is exported from the `@apollo/server-plugin-operation-registry` package.

Plugins with Mercurius

Some of the existing mercurius-specific Fastify plugins must be loaded after the mercurius plugin (read more here) on the plugin tree.

> warning **Warning** mercurius-upload is an exception and should be registered in the main file.

For this, `MercuriusDriver` exposes an optional `plugins` configuration option. It represents an array of objects that consist of two attributes: `plugin` and its `options`. Therefore, registering the cache plugin would look like this:

```
GraphQLModule.forRoot({
  driver: MercuriusDriver,
  // ...
  plugins: [
    {
      plugin: cache,
      options: {
        ttl: 10,
        policy: {
          Query: {
            add: true
          }
        }
      },
    }
  ]
}),
```

Complexity

> warning **Warning** This chapter applies only to the code first approach.

Query complexity allows you to define how complex certain fields are, and to restrict queries with a **maximum complexity**. The idea is to define how complex each field is by using a simple number. A common default is to give each field a complexity of `1`. In addition, the complexity calculation of a GraphQL query can be customized with so-called complexity estimators. A complexity estimator is a simple function that calculates the complexity for a field. You can add any number of complexity estimators to the rule, which are then executed one after another. The first estimator that returns a numeric complexity value determines the complexity for that field.

The `@nestjs/graphql` package integrates very well with tools like graphql-query-complexity that provides a cost analysis-based solution. With this library, you can reject queries to your GraphQL server that are deemed too costly to execute.

Installation

To begin using it, we first install the required dependency.

```
$ npm install --save graphql-query-complexity
```

Getting started

Once the installation process is complete, we can define the `ComplexityPlugin` class:

```
import { GraphQLSchemaHost } from "@nestjs/graphql";
import { Plugin } from "@nestjs/apollo";
import {
  ApolloServerPlugin,
  GraphQLRequestListener,
} from 'apollo-server-plugin-base';
import { GraphQLError } from 'graphql';
import {
  fieldExtensionsEstimator,
  getComplexity,
  simpleEstimator,
} from 'graphql-query-complexity';

@Plugin()
export class ComplexityPlugin implements ApolloServerPlugin {
  constructor(private gqlSchemaHost: GraphQLSchemaHost) {}

  async requestDidStart(): Promise<GraphQLRequestListener> {
    const maxComplexity = 20;
    const { schema } = this.gqlSchemaHost;

    return {
```

```
      async didResolveOperation({ request, document }) {
        const complexity = getComplexity({
          schema,
          operationName: request.operationName,
          query: document,
          variables: request.variables,
          estimators: [
            fieldExtensionsEstimator(),
            simpleEstimator({ defaultComplexity: 1 }),
          ],
        });
        if (complexity > maxComplexity) {
          throw new GraphQLError(
            `Query is too complex: ${complexity}. Maximum allowed
complexity: ${maxComplexity}`,
          );
        }
        console.log('Query Complexity:', complexity);
      },
    };
  }
}
```

For demonstration purposes, we specified the maximum allowed complexity as `20`. In the example above, we used 2 estimators, the `simpleEstimator` and the `fieldExtensionsEstimator`.

- `simpleEstimator`: the simple estimator returns a fixed complexity for each field
- `fieldExtensionsEstimator`: the field extensions estimator extracts the complexity value for each field of your schema

> info **Hint** Remember to add this class to the providers array in any module.

Field-level complexity

With this plugin in place, we can now define the complexity for any field by specifying the `complexity` property in the options object passed into the `@Field()` decorator, as follows:

```
@Field({ complexity: 3 })
title: string;
```

Alternatively, you can define the estimator function:

```
@Field({ complexity: (options: ComplexityEstimatorArgs) => ... })
title: string;
```

Query/Mutation-level complexity

In addition, `@Query()` and `@Mutation()` decorators may have a `complexity` property specified like so:

```
@Query({ complexity: (options: ComplexityEstimatorArgs) =>
options.args.count * options.childComplexity })
items(@Args('count') count: number) {
  return this.itemsService.getItems({ count });
}
```

Extensions

> warning **Warning** This chapter applies only to the code first approach.

Extensions is an **advanced, low-level feature** that lets you define arbitrary data in the types configuration. Attaching custom metadata to certain fields allows you to create more sophisticated, generic solutions. For example, with extensions, you can define field-level roles required to access particular fields. Such roles can be reflected at runtime to determine whether the caller has sufficient permissions to retrieve a specific field.

Adding custom metadata

To attach custom metadata for a field, use the `@Extensions()` decorator exported from the `@nestjs/graphql` package.

```
@Field()
@Extensions({ role: Role.ADMIN })
password: string;
```

In the example above, we assigned the `role` metadata property the value of `Role.ADMIN`. `Role` is a simple TypeScript enum that groups all the user roles available in our system.

Note, in addition to setting metadata on fields, you can use the `@Extensions()` decorator at the class level and method level (e.g., on the query handler).

Using custom metadata

Logic that leverages the custom metadata can be as complex as needed. For example, you can create a simple interceptor that stores/logs events per method invocation, or a field middleware that matches roles required to retrieve a field with the caller permissions (field-level permissions system).

For illustration purposes, let's define a `checkRoleMiddleware` that compares a user's role (hardcoded here) with a role required to access a target field:

```
export const checkRoleMiddleware: FieldMiddleware = async (
  ctx: MiddlewareContext,
  next: NextFn,
) => {
  const { info } = ctx;
  const { extensions } = info.parentType.getFields()[info.fieldName];

  /**
   * In a real-world application, the "userRole" variable
   * should represent the caller's (user) role (for example,
"ctx.user.role").
   */
  const userRole = Role.USER;
  if (userRole === extensions.role) {
```

```
    // or just "return null" to ignore
    throw new ForbiddenException(
      `User does not have sufficient permissions to access
"${info.fieldName}" field.`,
    );
  }
  return next();
};
```

With this in place, we can register a middleware for the `password` field, as follows:

```
@Field({ middleware: [checkRoleMiddleware] })
@Extensions({ role: Role.ADMIN })
password: string;
```

CLI Plugin

> warning **Warning** This chapter applies only to the code first approach.

TypeScript's metadata reflection system has several limitations which make it impossible to, for instance, determine what properties a class consists of or recognize whether a given property is optional or required. However, some of these constraints can be addressed at compilation time. Nest provides a plugin that enhances the TypeScript compilation process to reduce the amount of boilerplate code required.

> info **Hint** This plugin is **opt-in**. If you prefer, you can declare all decorators manually, or only specific decorators where you need them.

Overview

The GraphQL plugin will automatically:

- annotate all input object, object type and args classes properties with `@Field` unless `@HideField` is used
- set the `nullable` property depending on the question mark (e.g. `name?: string` will set `nullable: true`)
- set the `type` property depending on the type (supports arrays as well)
- generate descriptions for properties based on comments (if `introspectComments` set to `true`)

Please, note that your filenames **must have** one of the following suffixes in order to be analyzed by the plugin: `['.input.ts', '.args.ts', '.entity.ts', '.model.ts']` (e.g., `author.entity.ts`). If you are using a different suffix, you can adjust the plugin's behavior by specifying the `typeFileNameSuffix` option (see below).

With what we've learned so far, you have to duplicate a lot of code to let the package know how your type should be declared in GraphQL. For example, you could define a simple `Author` class as follows:

```
@@filename(authors/models/author.model)
@ObjectType()
export class Author {
  @Field(type => ID)
  id: number;

  @Field({ nullable: true })
  firstName?: string;

  @Field({ nullable: true })
  lastName?: string;

  @Field(type => [Post])
  posts: Post[];
}
```

While not a significant issue with medium-sized projects, it becomes verbose & hard to maintain once you have a large set of classes.

By enabling the GraphQL plugin, the above class definition can be declared simply:

```
@@filename(authors/models/author.model)
@ObjectType()
export class Author {
  @Field(type => ID)
  id: number;
  firstName?: string;
  lastName?: string;
  posts: Post[];
}
```

The plugin adds appropriate decorators on-the-fly based on the **Abstract Syntax Tree**. Thus, you won't have to struggle with `@Field` decorators scattered throughout the code.

> info **Hint** The plugin will automatically generate any missing GraphQL properties, but if you need to override them, simply set them explicitly via `@Field()`.

Comments introspection

With the comments introspection feature enabled, CLI plugin will generate descriptions for fields based on comments.

For example, given an example `roles` property:

```
/**
 * A list of user's roles
 */
@Field(() => [String], {
  description: `A list of user's roles`
})
roles: string[];
```

You must duplicate description values. With `introspectComments` enabled, the CLI plugin can extract these comments and automatically provide descriptions for properties. Now, the above field can be declared simply as follows:

```
/**
 * A list of user's roles
 */
roles: string[];
```

Using the CLI plugin

To enable the plugin, open `nest-cli.json` (if you use Nest CLI) and add the following `plugins` configuration:

```
{
  "collection": "@nestjs/schematics",
  "sourceRoot": "src",
  "compilerOptions": {
    "plugins": ["@nestjs/graphql"]
  }
}
```

You can use the `options` property to customize the behavior of the plugin.

```
"plugins": [
  {
    "name": "@nestjs/graphql",
    "options": {
      "typeFileNameSuffix": [".input.ts", ".args.ts"],
      "introspectComments": true
    }
  }
]
```

The `options` property has to fulfill the following interface:

```
export interface PluginOptions {
  typeFileNameSuffix?: string[];
  introspectComments?: boolean;
}
```

Option	Default	Description
`typeFileNameSuffix`	`['.input.ts', '.args.ts', '.entity.ts', '.model.ts']`	GraphQL types files suffix
`introspectComments`	`false`	If set to true, plugin will generate descriptions for properties based on comments

If you don't use the CLI but instead have a custom `webpack` configuration, you can use this plugin in combination with `ts-loader`:

```
getCustomTransformers: (program: any) => ({
  before: [require('@nestjs/graphql/plugin').before({}, program)]
}),
```

SWC builder

For standard setups (non-monorepo), to use CLI Plugins with the SWC builder, you need to enable type checking, as described here.

```
$ nest start -b swc --type-check
```

For monorepo setups, follow the instructions here.

```
$ npx ts-node src/generate-metadata.ts
# OR npx ts-node apps/{YOUR_APP}/src/generate-metadata.ts
```

Now, the serialized metadata file must be loaded by the `GraphQLModule` method, as shown below:

```
import metadata from './metadata'; // <-- file auto-generated by the
"PluginMetadataGenerator"

GraphQLModule.forRoot<...>({
  ..., // other options
  metadata,
}),
```

Integration with `ts-jest` (e2e tests)

When running e2e tests with this plugin enabled, you may run into issues with compiling schema. For example, one of the most common errors is:

```
Object type <name> must define one or more fields.
```

This happens because `jest` configuration does not import `@nestjs/graphql/plugin` plugin anywhere.

To fix this, create the following file in your e2e tests directory:

```
const transformer = require('@nestjs/graphql/plugin');

module.exports.name = 'nestjs-graphql-transformer';
// you should change the version number anytime you change the
configuration below - otherwise, jest will not detect changes
module.exports.version = 1;

module.exports.factory = (cs) => {
  return transformer.before(
    {
```

```
      // @nestjs/graphql/plugin options (can be empty)
    },
    cs.program, // "cs.tsCompiler.program" for older versions of Jest (<=
v27)
  );
};
```

With this in place, import AST transformer within your `jest` configuration file. By default (in the starter application), e2e tests configuration file is located under the `test` folder and is named `jest-e2e.json`.

```
{
  ... // other configuration
  "globals": {
    "ts-jest": {
      "astTransformers": {
        "before": ["<path to the file created above>"]
      }
    }
  }
}
```

If you use `jest@^29`, then use the snippet below, as the previous approach got deprecated.

```
{
  ... // other configuration
  "transform": {
    "^.+\\.(t|j)s$": [
      "ts-jest",
      {
        "astTransformers": {
          "before": ["<path to the file created above>"]
        }
      }
    ]
  }
}
```

Sharing models

> warning **Warning** This chapter applies only to the code first approach.

One of the biggest advantages of using Typescript for the backend of your project is the ability to reuse the same models in a Typescript-based frontend application, by using a common Typescript package.

But there's a problem: the models created using the code first approach are heavily decorated with GraphQL related decorators. Those decorators are irrelevant in the frontend, negatively impacting performance.

Using the model shim

To solve this issue, NestJS provides a "shim" which allows you to replace the original decorators with inert code by using a `webpack` (or similar) configuration. To use this shim, configure an alias between the `@nestjs/graphql` package and the shim.

For example, for webpack this is resolved this way:

```
resolve: { // see: https://webpack.js.org/configuration/resolve/
  alias: {
      "@nestjs/graphql": path.resolve(__dirname,
"../node_modules/@nestjs/graphql/dist/extra/graphql-model-shim")
  }
}
```

> info **Hint** The TypeORM package has a similar shim that can be found here.

Other features

In the GraphQL world, there is a lot of debate about handling issues like **authentication**, or **side-effects** of operations. Should we handle things inside the business logic? Should we use a higher-order function to enhance queries and mutations with authorization logic? Or should we use schema directives? There is no single one-size-fits-all answer to these questions.

Nest helps address these issues with its cross-platform features like guards and interceptors. The philosophy is to reduce redundancy and provide tooling that helps create well-structured, readable, and consistent applications.

Overview

You can use standard guards, interceptors, filters and pipes in the same fashion with GraphQL as with any RESTful application. Additionally, you can easily create your own decorators by leveraging the custom decorators feature. Let's take a look at a sample GraphQL query handler.

```
@Query('author')
@UseGuards(AuthGuard)
async getAuthor(@Args('id', ParseIntPipe) id: number) {
  return this.authorsService.findOneById(id);
}
```

As you can see, GraphQL works with both guards and pipes in the same way as HTTP REST handlers. Because of this, you can move your authentication logic to a guard; you can even reuse the same guard class across both a REST and GraphQL API interface. Similarly, interceptors work across both types of applications in the same way:

```
@Mutation()
@UseInterceptors(EventsInterceptor)
async upvotePost(@Args('postId') postId: number) {
  return this.postsService.upvoteById({ id: postId });
}
```

Execution context

Since GraphQL receives a different type of data in the incoming request, the execution context received by both guards and interceptors is somewhat different with GraphQL vs. REST. GraphQL resolvers have a distinct set of arguments: `root`, `args`, `context`, and `info`. Thus guards and interceptors must transform the generic `ExecutionContext` to a `GqlExecutionContext`. This is straightforward:

```
import { CanActivate, ExecutionContext, Injectable } from
'@nestjs/common';
import { GqlExecutionContext } from '@nestjs/graphql';
```

```
@Injectable()
export class AuthGuard implements CanActivate {
  canActivate(context: ExecutionContext): boolean {
    const ctx = GqlExecutionContext.create(context);
    return true;
  }
}
```

The GraphQL context object returned by `GqlExecutionContext.create()` exposes a **get** method for each GraphQL resolver argument (e.g., `getArgs()`, `getContext()`, etc). Once transformed, we can easily pick out any GraphQL argument for the current request.

Exception filters

Nest standard exception filters are compatible with GraphQL applications as well. As with `ExecutionContext`, GraphQL apps should transform the `ArgumentsHost` object to a `GqlArgumentsHost` object.

```
@Catch(HttpException)
export class HttpExceptionFilter implements GqlExceptionFilter {
  catch(exception: HttpException, host: ArgumentsHost) {
    const gqlHost = GqlArgumentsHost.create(host);
    return exception;
  }
}
```

> info **Hint** Both `GqlExceptionFilter` and `GqlArgumentsHost` are imported from the `@nestjs/graphql` package.

Note that unlike the REST case, you don't use the native `response` object to generate a response.

Custom decorators

As mentioned, the custom decorators feature works as expected with GraphQL resolvers.

```
export const User = createParamDecorator(
  (data: unknown, ctx: ExecutionContext) =>
    GqlExecutionContext.create(ctx).getContext().user,
);
```

Use the `@User()` custom decorator as follows:

```
@Mutation()
async upvotePost(
  @User() user: UserEntity,
```

```
    @Args('postId') postId: number,
  ) {}
```

> info **Hint** In the above example, we have assumed that the `user` object is assigned to the context of your GraphQL application.

Execute enhancers at the field resolver level

In the GraphQL context, Nest does not run **enhancers** (the generic name for interceptors, guards and filters) at the field level see this issue: they only run for the top level `@Query()`/`@Mutation()` method. You can tell Nest to execute interceptors, guards or filters for methods annotated with `@ResolveField()` by setting the `fieldResolverEnhancers` option in `GqlModuleOptions`. Pass it a list of `'interceptors'`, `'guards'`, and/or `'filters'` as appropriate:

```
GraphQLModule.forRoot({
  fieldResolverEnhancers: ['interceptors']
}),
```

> **Warning** Enabling enhancers for field resolvers can cause performance issues when you are returning lots of records and your field resolver is executed thousands of times. For this reason, when you enable `fieldResolverEnhancers`, we advise you to skip execution of enhancers that are not strictly necessary for your field resolvers. You can do this using the following helper function:

```
export function isResolvingGraphQLField(context: ExecutionContext):
boolean {
  if (context.getType<GqlContextType>() === 'graphql') {
    const gqlContext = GqlExecutionContext.create(context);
    const info = gqlContext.getInfo();
    const parentType = info.parentType.name;
    return parentType !== 'Query' && parentType !== 'Mutation';
  }
  return false;
}
```

Creating a custom driver

Nest provides two official drivers out-of-the-box: `@nestjs/apollo` and `@nestjs/mercurius`, as well as an API allowing developers to build new **custom drivers**. With a custom driver, you can integrate any GraphQL library or extend the existing integration, adding extra features on top.

For example, to integrate the `express-graphql` package, you could create the following driver class:

```
import { AbstractGraphQLDriver, GqlModuleOptions } from '@nestjs/graphql';
import { graphqlHTTP } from 'express-graphql';
```

```
class ExpressGraphQLDriver extends AbstractGraphQLDriver {
  async start(options: GqlModuleOptions<any>): Promise<void> {
    options = await this.graphQlFactory.mergeWithSchema(options);

    const { httpAdapter } = this.httpAdapterHost;
    httpAdapter.use(
      '/graphql',
      graphqlHTTP({
        schema: options.schema,
        graphiql: true,
      }),
    );
  }

  async stop() {}
}
```

And then use it as follows:

```
GraphQLModule.forRoot({
  driver: ExpressGraphQLDriver,
});
```

Federation

Federation offers a means of splitting your monolithic GraphQL server into independent microservices. It consists of two components: a gateway and one or more federated microservices. Each microservice holds part of the schema and the gateway merges the schemas into a single schema that can be consumed by the client.

To quote the Apollo docs, Federation is designed with these core principles:

- Building a graph should be **declarative.** With federation, you compose a graph declaratively from within your schema instead of writing imperative schema stitching code.
- Code should be separated by **concern**, not by types. Often no single team controls every aspect of an important type like a User or Product, so the definition of these types should be distributed across teams and codebases, rather than centralized.
- The graph should be simple for clients to consume. Together, federated services can form a complete, product-focused graph that accurately reflects how it's being consumed on the client.
- It's just **GraphQL**, using only spec-compliant features of the language. Any language, not just JavaScript, can implement federation.

> warning **Warning** Federation currently does not support subscriptions.

In the following sections, we'll set up a demo application that consists of a gateway and two federated endpoints: Users service and Posts service.

Federation with Apollo

Start by installing the required dependencies:

```
$ npm install --save @apollo/federation @apollo/subgraph
```

Schema first

The "User service" provides a simple schema. Note the `@key` directive: it instructs the Apollo query planner that a particular instance of `User` can be fetched if you specify its `id`. Also, note that we `extend` the `Query` type.

```
type User @key(fields: "id") {
  id: ID!
  name: String!
}

extend type Query {
  getUser(id: ID!): User
}
```

Resolver provides one additional method named `resolveReference()`. This method is triggered by the Apollo Gateway whenever a related resource requires a User instance. We'll see an example of this in the Posts service later. Please note that the method must be annotated with the `@ResolveReference()` decorator.

```
import { Args, Query, Resolver, ResolveReference } from '@nestjs/graphql';
import { UsersService } from './users.service';

@Resolver('User')
export class UsersResolver {
  constructor(private usersService: UsersService) {}

  @Query()
  getUser(@Args('id') id: string) {
    return this.usersService.findById(id);
  }

  @ResolveReference()
  resolveReference(reference: { __typename: string; id: string }) {
    return this.usersService.findById(reference.id);
  }
}
```

Finally, we hook everything up by registering the `GraphQLModule` passing the `ApolloFederationDriver` driver in the configuration object:

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
} from '@nestjs/apollo';
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';
import { UsersResolver } from './users.resolver';

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      typePaths: ['**/*.graphql'],
    }),
  ],
  providers: [UsersResolver],
})
export class AppModule {}
```

Code first

Start by adding some extra decorators to the `User` entity.

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

@ObjectType()
@Directive('@key(fields: "id")')
export class User {
  @Field((type) => ID)
  id: number;

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

Resolver provides one additional method named `resolveReference()`. This method is triggered by the Apollo Gateway whenever a related resource requires a User instance. We'll see an example of this in the Posts service later. Please note that the method must be annotated with the `@ResolveReference()` decorator.

```
import { Args, Query, Resolver, ResolveReference } from '@nestjs/graphql';
import { User } from './user.entity';
import { UsersService } from './users.service';

@Resolver((of) => User)
export class UsersResolver {
  constructor(private usersService: UsersService) {}

  @Query((returns) => User)
  getUser(@Args('id') id: number): User {
    return this.usersService.findById(id);
  }

  @ResolveReference()
  resolveReference(reference: { __typename: string; id: number }): User {
    return this.usersService.findById(reference.id);
  }
}
```

Finally, we hook everything up by registering the `GraphQLModule` passing the `ApolloFederationDriver` driver in the configuration object:

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
} from '@nestjs/apollo';
import { Module } from '@nestjs/common';
import { UsersResolver } from './users.resolver';
import { UsersService } from './users.service'; // Not included in this
example
```

```
@Module({
  imports: [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      autoSchemaFile: true,
    }),
  ],
  providers: [UsersResolver, UsersService],
})
export class AppModule {}
```

A working example is available here in code first mode and here in schema first mode.

Federated example: Posts

Post service is supposed to serve aggregated posts through the `getPosts` query, but also extend our `User` type with the `user.posts` field.

Schema first

"Posts service" references the `User` type in its schema by marking it with the `extend` keyword. It also declares one additional property on the `User` type (`posts`). Note the `@key` directive used for matching instances of User, and the `@external` directive indicating that the `id` field is managed elsewhere.

```
type Post @key(fields: "id") {
  id: ID!
  title: String!
  body: String!
  user: User
}

extend type User @key(fields: "id") {
  id: ID! @external
  posts: [Post]
}

extend type Query {
  getPosts: [Post]
}
```

In the following example, the `PostsResolver` provides the `getUser()` method that returns a reference containing `__typename` and some additional properties your application may need to resolve the reference, in this case `id`. `__typename` is used by the GraphQL Gateway to pinpoint the microservice responsible for the User type and retrieve the corresponding instance. The "Users service" described above will be requested upon execution of the `resolveReference()` method.

```
import { Query, Resolver, Parent, ResolveField } from '@nestjs/graphql';
import { PostsService } from './posts.service';
import { Post } from './posts.interfaces';

@Resolver('Post')
export class PostsResolver {
  constructor(private postsService: PostsService) {}

  @Query('getPosts')
  getPosts() {
    return this.postsService.findAll();
  }

  @ResolveField('user')
  getUser(@Parent() post: Post) {
    return { __typename: 'User', id: post.userId };
  }
}
```

Lastly, we must register the `GraphQLModule`, similarly to what we did in the "Users service" section.

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
} from '@nestjs/apollo';
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';
import { PostsResolver } from './posts.resolver';

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      typePaths: ['**/*.graphql'],
    }),
  ],
  providers: [PostsResolvers],
})
export class AppModule {}
```

Code first

First, we will have to declare a class representing the `User` entity. Although the entity itself lives in another service, we will be using it (extending its definition) here. Note the `@extends` and `@external` directives.

```
import { Directive, ObjectType, Field, ID } from '@nestjs/graphql';
import { Post } from './post.entity';
```

```
@ObjectType()
@Directive('@extends')
@Directive('@key(fields: "id")')
export class User {
  @Field((type) => ID)
  @Directive('@external')
  id: number;

  @Field((type) => [Post])
  posts?: Post[];
}
```

Now let's create the corresponding resolver for our extension on the `User` entity, as follows:

```
import { Parent, ResolveField, Resolver } from '@nestjs/graphql';
import { PostsService } from './posts.service';
import { Post } from './post.entity';
import { User } from './user.entity';

@Resolver((of) => User)
export class UsersResolver {
  constructor(private readonly postsService: PostsService) {}

  @ResolveField((of) => [Post])
  public posts(@Parent() user: User): Post[] {
    return this.postsService.forAuthor(user.id);
  }
}
```

We also have to define the `Post` entity class:

```
import { Directive, Field, ID, Int, ObjectType } from '@nestjs/graphql';
import { User } from './user.entity';

@ObjectType()
@Directive('@key(fields: "id")')
export class Post {
  @Field((type) => ID)
  id: number;

  @Field()
  title: string;

  @Field((type) => Int)
  authorId: number;

  @Field((type) => User)
  user?: User;
}
```

And its resolver:

```
import { Query, Args, ResolveField, Resolver, Parent } from
'@nestjs/graphql';
import { PostsService } from './posts.service';
import { Post } from './post.entity';
import { User } from './user.entity';

@Resolver((of) => Post)
export class PostsResolver {
  constructor(private readonly postsService: PostsService) {}

  @Query((returns) => Post)
  findPost(@Args('id') id: number): Post {
    return this.postsService.findOne(id);
  }

  @Query((returns) => [Post])
  getPosts(): Post[] {
    return this.postsService.all();
  }

  @ResolveField((of) => User)
  user(@Parent() post: Post): any {
    return { __typename: 'User', id: post.authorId };
  }
}
```

And finally, tie it together in a module. Note the schema build options, where we specify that `User` is an orphaned (external) type.

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
} from '@nestjs/apollo';
import { Module } from '@nestjs/common';
import { User } from './user.entity';
import { PostsResolvers } from './posts.resolvers';
import { UsersResolvers } from './users.resolvers';
import { PostsService } from './posts.service'; // Not included in example

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      autoSchemaFile: true,
      buildSchemaOptions: {
        orphanedTypes: [User],
      },
```

```
    }),
  ],
  providers: [PostsResolver, UsersResolver, PostsService],
})
export class AppModule {}
```

A working example is available here for the code first mode and here for the schema first mode.

Federated example: Gateway

Start by installing the required dependency:

```
$ npm install --save @apollo/gateway
```

The gateway requires a list of endpoints to be specified and it will auto-discover the corresponding schemas. Therefore the implementation of the gateway service will remain the same for both code and schema first approaches.

```
import { IntrospectAndCompose } from '@apollo/gateway';
import { ApolloGatewayDriver, ApolloGatewayDriverConfig } from
'@nestjs/apollo';
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloGatewayDriverConfig>({
      driver: ApolloGatewayDriver,
      server: {
        // ... Apollo server options
        cors: true,
      },
      gateway: {
        supergraphSdl: new IntrospectAndCompose({
          subgraphs: [
            { name: 'users', url: 'http://user-service/graphql' },
            { name: 'posts', url: 'http://post-service/graphql' },
          ],
        }),
      },
    }),
  ],
})
export class AppModule {}
```

A working example is available here for the code first mode and here for the schema first mode.

Federation with Mercurius

Start by installing the required dependencies:

```
$ npm install --save @apollo/subgraph @nestjs/mercurius
```

> info **Note** The `@apollo/subgraph` package is required to build a subgraph schema (`buildSubgraphSchema`, `printSubgraphSchema` functions).

Schema first

The "User service" provides a simple schema. Note the `@key` directive: it instructs the Mercurius query planner that a particular instance of `User` can be fetched if you specify its `id`. Also, note that we `extend` the `Query` type.

```
type User @key(fields: "id") {
  id: ID!
  name: String!
}

extend type Query {
  getUser(id: ID!): User
}
```

Resolver provides one additional method named `resolveReference()`. This method is triggered by the Mercurius Gateway whenever a related resource requires a User instance. We'll see an example of this in the Posts service later. Please note that the method must be annotated with the `@ResolveReference()` decorator.

```
import { Args, Query, Resolver, ResolveReference } from '@nestjs/graphql';
import { UsersService } from './users.service';

@Resolver('User')
export class UsersResolver {
  constructor(private usersService: UsersService) {}

  @Query()
  getUser(@Args('id') id: string) {
    return this.usersService.findById(id);
  }

  @ResolveReference()
  resolveReference(reference: { __typename: string; id: string }) {
    return this.usersService.findById(reference.id);
  }
}
```

Finally, we hook everything up by registering the `GraphQLModule` passing the `MercuriusFederationDriver` driver in the configuration object:

```
import {
  MercuriusFederationDriver,
  MercuriusFederationDriverConfig,
} from '@nestjs/mercurius';
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';
import { UsersResolver } from './users.resolver';

@Module({
  imports: [
    GraphQLModule.forRoot<MercuriusFederationDriverConfig>({
      driver: MercuriusFederationDriver,
      typePaths: ['**/*.graphql'],
      federationMetadata: true,
    }),
  ],
  providers: [UsersResolver],
})
export class AppModule {}
```

Code first

Start by adding some extra decorators to the `User` entity.

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

@ObjectType()
@Directive('@key(fields: "id")')
export class User {
  @Field((type) => ID)
  id: number;

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

Resolver provides one additional method named `resolveReference()`. This method is triggered by the Mercurius Gateway whenever a related resource requires a User instance. We'll see an example of this in the Posts service later. Please note that the method must be annotated with the `@ResolveReference()` decorator.

```
import { Args, Query, Resolver, ResolveReference } from '@nestjs/graphql';
import { User } from './user.entity';
import { UsersService } from './users.service';
```

```
@Resolver((of) => User)
export class UsersResolver {
  constructor(private usersService: UsersService) {}

  @Query((returns) => User)
  getUser(@Args('id') id: number): User {
    return this.usersService.findById(id);
  }

  @ResolveReference()
  resolveReference(reference: { __typename: string; id: number }): User {
    return this.usersService.findById(reference.id);
  }
}
```

Finally, we hook everything up by registering the `GraphQLModule` passing the `MercuriusFederationDriver` driver in the configuration object:

```
import {
  MercuriusFederationDriver,
  MercuriusFederationDriverConfig,
} from '@nestjs/mercurius';
import { Module } from '@nestjs/common';
import { UsersResolver } from './users.resolver';
import { UsersService } from './users.service'; // Not included in this
example

@Module({
  imports: [
    GraphQLModule.forRoot<MercuriusFederationDriverConfig>({
      driver: MercuriusFederationDriver,
      autoSchemaFile: true,
      federationMetadata: true,
    }),
  ],
  providers: [UsersResolver, UsersService],
})
export class AppModule {}
```

Federated example: Posts

Post service is supposed to serve aggregated posts through the `getPosts` query, but also extend our `User` type with the `user.posts` field.

Schema first

"Posts service" references the `User` type in its schema by marking it with the `extend` keyword. It also declares one additional property on the `User` type (`posts`). Note the `@key` directive used for matching

instances of User, and the `@external` directive indicating that the `id` field is managed elsewhere.

```
type Post @key(fields: "id") {
  id: ID!
  title: String!
  body: String!
  user: User
}

extend type User @key(fields: "id") {
  id: ID! @external
  posts: [Post]
}

extend type Query {
  getPosts: [Post]
}
```

In the following example, the `PostsResolver` provides the `getUser()` method that returns a reference containing `__typename` and some additional properties your application may need to resolve the reference, in this case `id`. `__typename` is used by the GraphQL Gateway to pinpoint the microservice responsible for the User type and retrieve the corresponding instance. The "Users service" described above will be requested upon execution of the `resolveReference()` method.

```
import { Query, Resolver, Parent, ResolveField } from '@nestjs/graphql';
import { PostsService } from './posts.service';
import { Post } from './posts.interfaces';

@Resolver('Post')
export class PostsResolver {
  constructor(private postsService: PostsService) {}

  @Query('getPosts')
  getPosts() {
    return this.postsService.findAll();
  }

  @ResolveField('user')
  getUser(@Parent() post: Post) {
    return { __typename: 'User', id: post.userId };
  }
}
```

Lastly, we must register the `GraphQLModule`, similarly to what we did in the "Users service" section.

```
import {
  MercuriusFederationDriver,
  MercuriusFederationDriverConfig,
```

```
} from '@nestjs/mercurius';
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';
import { PostsResolver } from './posts.resolver';

@Module({
  imports: [
    GraphQLModule.forRoot<MercuriusFederationDriverConfig>({
      driver: MercuriusFederationDriver,
      federationMetadata: true,
      typePaths: ['**/*.graphql'],
    }),
  ],
  providers: [PostsResolvers],
})
export class AppModule {}
```

Code first

First, we will have to declare a class representing the `User` entity. Although the entity itself lives in another service, we will be using it (extending its definition) here. Note the `@extends` and `@external` directives.

```
import { Directive, ObjectType, Field, ID } from '@nestjs/graphql';
import { Post } from './post.entity';

@ObjectType()
@Directive('@extends')
@Directive('@key(fields: "id")')
export class User {
  @Field((type) => ID)
  @Directive('@external')
  id: number;

  @Field((type) => [Post])
  posts?: Post[];
}
```

Now let's create the corresponding resolver for our extension on the `User` entity, as follows:

```
import { Parent, ResolveField, Resolver } from '@nestjs/graphql';
import { PostsService } from './posts.service';
import { Post } from './post.entity';
import { User } from './user.entity';

@Resolver((of) => User)
export class UsersResolver {
  constructor(private readonly postsService: PostsService) {}

  @ResolveField((of) => [Post])
```

```
  public posts(@Parent() user: User): Post[] {
    return this.postsService.forAuthor(user.id);
  }
}
```

We also have to define the `Post` entity class:

```
import { Directive, Field, ID, Int, ObjectType } from '@nestjs/graphql';
import { User } from './user.entity';

@ObjectType()
@Directive('@key(fields: "id")')
export class Post {
  @Field((type) => ID)
  id: number;

  @Field()
  title: string;

  @Field((type) => Int)
  authorId: number;

  @Field((type) => User)
  user?: User;
}
```

And its resolver:

```
import { Query, Args, ResolveField, Resolver, Parent } from
'@nestjs/graphql';
import { PostsService } from './posts.service';
import { Post } from './post.entity';
import { User } from './user.entity';

@Resolver((of) => Post)
export class PostsResolver {
  constructor(private readonly postsService: PostsService) {}

  @Query((returns) => Post)
  findPost(@Args('id') id: number): Post {
    return this.postsService.findOne(id);
  }

  @Query((returns) => [Post])
  getPosts(): Post[] {
    return this.postsService.all();
  }

  @ResolveField((of) => User)
```

```
  user(@Parent() post: Post): any {
    return { __typename: 'User', id: post.authorId };
  }
}
```

And finally, tie it together in a module. Note the schema build options, where we specify that `User` is an orphaned (external) type.

```
import {
  MercuriusFederationDriver,
  MercuriusFederationDriverConfig,
} from '@nestjs/mercurius';
import { Module } from '@nestjs/common';
import { User } from './user.entity';
import { PostsResolvers } from './posts.resolvers';
import { UsersResolvers } from './users.resolvers';
import { PostsService } from './posts.service'; // Not included in example

@Module({
  imports: [
    GraphQLModule.forRoot<MercuriusFederationDriverConfig>({
      driver: MercuriusFederationDriver,
      autoSchemaFile: true,
      federationMetadata: true,
      buildSchemaOptions: {
        orphanedTypes: [User],
      },
    }),
  ],
  providers: [PostsResolver, UsersResolver, PostsService],
})
export class AppModule {}
```

Federated example: Gateway

The gateway requires a list of endpoints to be specified and it will auto-discover the corresponding schemas. Therefore the implementation of the gateway service will remain the same for both code and schema first approaches.

```
import {
  MercuriusGatewayDriver,
  MercuriusGatewayDriverConfig,
} from '@nestjs/mercurius';
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';

@Module({
  imports: [
```

```
    GraphQLModule.forRoot<MercuriusGatewayDriverConfig>({
      driver: MercuriusGatewayDriver,
      gateway: {
        services: [
          { name: 'users', url: 'http://user-service/graphql' },
          { name: 'posts', url: 'http://post-service/graphql' },
        ],
      },
    }),
  ],
})
export class AppModule {}
```

Federation 2

To quote the Apollo docs, Federation 2 improves developer experience from the original Apollo Federation (called Federation 1 in this doc), which is backward compatible with most original supergraphs.

> warning **Warning** Mercurius doesn't fully support Federation 2. You can see the list of libraries that support Federation 2 here.

In the following sections, we'll upgrade the previous example to Federation 2.

Federated example: Users

One change in Federation 2 is that entities have no originating subgraph, so we don't need to extend `Query` anymore. For more detail please refer to the entities topic in Apollo Federation 2 docs.

Schema first

We can simply remove `extend` keyword from the schema.

```
type User @key(fields: "id") {
  id: ID!
  name: String!
}

type Query {
  getUser(id: ID!): User
}
```

Code first

To use Federation 2, we need to specify the federation version in `autoSchemaFile` option.

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
```

```
} from '@nestjs/apollo';
import { Module } from '@nestjs/common';
import { UsersResolver } from './users.resolver';
import { UsersService } from './users.service'; // Not included in this
example

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      autoSchemaFile: {
        federation: 2,
      },
    }),
  ],
  providers: [UsersResolver, UsersService],
})
export class AppModule {}
```

Federated example: Posts

With the same reason as above, we don't need to extend `User` and `Query` anymore.

Schema first

We can simply remove `extend` and `external` directives from the schema

```
type Post @key(fields: "id") {
  id: ID!
  title: String!
  body: String!
  user: User
}

type User @key(fields: "id") {
  id: ID!
  posts: [Post]
}

type Query {
  getPosts: [Post]
}
```

Code first

Since we don't extend `User` entity anymore, we can simply remove `extends` and `external` directives from `User`.

```
import { Directive, ObjectType, Field, ID } from '@nestjs/graphql';
import { Post } from './post.entity';

@ObjectType()
@Directive('@key(fields: "id")')
export class User {
  @Field((type) => ID)
  id: number;

  @Field((type) => [Post])
  posts?: Post[];
}
```

Also, similarly to the User service, we need to specify in the `GraphQLModule` to use Federation 2.

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
} from '@nestjs/apollo';
import { Module } from '@nestjs/common';
import { User } from './user.entity';
import { PostsResolvers } from './posts.resolvers';
import { UsersResolvers } from './users.resolvers';
import { PostsService } from './posts.service'; // Not included in example

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      autoSchemaFile: {
        federation: 2,
      },
      buildSchemaOptions: {
        orphanedTypes: [User],
      },
    }),
  ],
  providers: [PostsResolver, UsersResolver, PostsService],
})
export class AppModule {}
```

Migrating to v11 from v10

This chapter provides a set of guidelines for migrating from `@nestjs/graphql` version 10 to version 11. As part of this major release, we updated the Apollo driver to be compatible with Apollo Server v4 (instead of v3). Note: there are several breaking changes in Apollo Server v4 (especially around plugins and ecosystem packages), so you'll have to update your codebase accordingly. For more information, see the Apollo Server v4 migration guide.

Apollo packages

Instead of installing the `apollo-server-express` package, you'll have to install `@apollo/server`:

```
$ npm uninstall apollo-server-express
$ npm install @apollo/server
```

If you use the Fastify adapter, you'll have to install the `@as-integrations/fastify` package instead:

```
$ npm uninstall apollo-server-fastify
$ npm install @apollo/server @as-integrations/fastify
```

Mercurius packages

Mercurius gateway is no longer a part of the `mercurius` package. Instead, you'll have to install the `@mercuriusjs/gateway` package separately:

```
$ npm install @mercuriusjs/gateway
```

Similarly, for creating federated schemas, you'll have to install the `@mercuriusjs/federation` package:

```
$ npm install @mercuriusjs/federation
```

Migrating to v10 from v9

This chapter provides a set of guidelines for migrating from `@nestjs/graphql` version 9 to version 10. The focus of this major-version release is to provide a lighter, platform-agnostic core library.

Introducing "driver" packages

In the latest version, we made a decision to break the `@nestjs/graphql` package up into a few separate libraries, letting you choose whether to use Apollo (`@nestjs/apollo`), Mercurius (`@nestjs/mercurius`), or another GraphQL library in your project.

This implies that now you have to explicitly specify what driver your application will use.

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

@Module({
  imports: [
    GraphQLModule.forRoot({
      autoSchemaFile: 'schema.gql',
    }),
  ],
})
export class AppModule {}

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

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloDriverConfig>({
      driver: ApolloDriver,
      autoSchemaFile: 'schema.gql',
    }),
  ],
})
export class AppModule {}
```

Plugins

Apollo Server plugins let you perform custom operations in response to certain events. Since this is an exclusive Apollo feature, we moved it from the `@nestjs/graphql` to the newly created `@nestjs/apollo` package so you'll have to update imports in your application.

```
// Before
import { Plugin } from '@nestjs/graphql';

// After
import { Plugin } from '@nestjs/apollo';
```

Directives

`schemaDirectives` feature has been replaced with the new Schema directives API in v8 of `@graphql-tools/schema` package.

```
// Before
import { SchemaDirectiveVisitor } from '@graphql-tools/utils';
import { defaultFieldResolver, GraphQLField } from 'graphql';

export class UpperCaseDirective extends SchemaDirectiveVisitor {
  visitFieldDefinition(field: GraphQLField<any, any>) {
    const { resolve = defaultFieldResolver } = field;
    field.resolve = async function (...args) {
      const result = await resolve.apply(this, args);
      if (typeof result === 'string') {
        return result.toUpperCase();
      }
      return result;
    };
  }
}

// After
import { getDirective, MapperKind, mapSchema } from '@graphql-
tools/utils';
import { defaultFieldResolver, GraphQLSchema } from 'graphql';

export function upperDirectiveTransformer(
  schema: GraphQLSchema,
  directiveName: string,
) {
  return mapSchema(schema, {
    [MapperKind.OBJECT_FIELD]: (fieldConfig) => {
      const upperDirective = getDirective(
        schema,
        fieldConfig,
        directiveName,
      )?.[0];

      if (upperDirective) {
        const { resolve = defaultFieldResolver } = fieldConfig;

        // Replace the original resolver with a function that *first*
calls
        // the original resolver, then converts its result to upper case
        fieldConfig.resolve = async function (source, args, context, info)
{
          const result = await resolve(source, args, context, info);
          if (typeof result === 'string') {
            return result.toUpperCase();
          }
          return result;
        };
        return fieldConfig;
      }
    },
  });
}
```

To apply this directive implementation to a schema that contains `@upper` directives, use the `transformSchema` function:

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  ...
  transformSchema: schema => upperDirectiveTransformer(schema, 'upper'),
})
```

Federation

`GraphQLFederationModule` has been removed and replaced with the corresponding driver class:

```
// Before
GraphQLFederationModule.forRoot({
  autoSchemaFile: true,
});

// After
GraphQLModule.forRoot<ApolloFederationDriverConfig>({
  driver: ApolloFederationDriver,
  autoSchemaFile: true,
});
```

> info **Hint** Both `ApolloFederationDriver` class and `ApolloFederationDriverConfig` are exported from the `@nestjs/apollo` package.

Likewise, instead of using a dedicated `GraphQLGatewayModule`, simply pass the appropriate `driver` class to your `GraphQLModule` settings:

```
// Before
GraphQLGatewayModule.forRoot({
  gateway: {
    supergraphSdl: new IntrospectAndCompose({
      subgraphs: [
        { name: 'users', url: 'http://localhost:3000/graphql' },
        { name: 'posts', url: 'http://localhost:3001/graphql' },
      ],
    }),
  },
});

// After
GraphQLModule.forRoot<ApolloGatewayDriverConfig>({
  driver: ApolloGatewayDriver,
  gateway: {
    supergraphSdl: new IntrospectAndCompose({
```

```
        subgraphs: [
          { name: 'users', url: 'http://localhost:3000/graphql' },
          { name: 'posts', url: 'http://localhost:3001/graphql' },
        ],
      }),
    },
  });
```

> info **Hint** Both `ApolloGatewayDriver` class and `ApolloGatewayDriverConfig` are exported from the `@nestjs/apollo` package.