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.