Resolvers

Typically, you have to create a resolvers map manually. The `@nestjs/graphql` package, on the other hand, generate resolvers map automatically using the metadata provided by the decorators. In order to learn the library basics, we'll create a simple authors API.

Schema first

As mentioned in the **previous** chapter, in the schema first approach we have to manually define our types in SDL (read **more**).

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

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

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

Our GraphQL schema contains single query exposed - `author(id: Int!): Author`. Now, let's create an `AuthorResolver`.

```
@Resolver('Author')
export class AuthorResolver {
  constructor(
    private readonly authorsService: AuthorsService,
    private readonly postsService: PostsService,
  ) {}

  @Query()
  async author(@Args('id') id: number) {
```

```
    return await this.authorsService.findOneById(id);
  }

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

> **HINT**
> If you use the `@Resolver()` decorator, you don't have to mark a class as an `@Injectable()`, otherwise, it's necessary.

The `@Resolver()` decorator does not affect queries and mutations (neither `@Query()` nor `@Mutation()` decorators). It only informs Nest that each `@ResolveProperty()` inside this particular class has a parent, which is an `Author` type in this case (`Author.posts` relation). Basically, instead of setting `@Resolver()` at the top of the class, this can be done close to the method:

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

However, if you have multiple `@ResolveProperty()` inside one class, you would have to add `@Resolver()` to all of them which is not necessarily a good practice (creates an extra overhead).

Conventionally, we would use something like `getAuthor()` or `getPosts()` as method names. We can easily do this by moving the real names between the parentheses of the decorator.

```
@Resolver('Author')
export class AuthorResolver {
  constructor(
    private readonly authorsService: AuthorsService,
    private readonly postsService: PostsService,
  ) {}

  @Query('author')
  async getAuthor(@Args('id') id: number) {
```

```
    return await this.authorsService.findOneById(id);
  }

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

> **HINT**
> The `@Resolver()` decorator can be used at the method-level as well.

Typings

Assuming that we have enabled the typings generation feature (with `outputAs: 'class'`) in the **previous** chapter, once you run our application it should generate the following file:

```
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>;
}
```

Classes allow you using **decorators** which makes them extremely useful in terms of the validation purposes (read **more**). For example:

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

export class CreatePostInput {
```

```
  @MinLength(3)
  @MaxLength(50)
  title: string;
}
```

> **NOTICE**
> To enable auto-validation of your inputs (and parameters), you have to use `ValidationPipe`. Read more about validation here or more specifically about pipes here.

Nonetheless, if you add your decorators directly into the automatically generated file, they will be **thrown away** on each consecutive change. Hence, you should rather 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;
}
```

Code first

In the code first approach, we don't have to write SDL by hand. Instead we'll only use decorators.

```
import { Field, Int, ObjectType } from 'type-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[];
```

```
}
```

`Author` model has been created. Now, let's create the missing `Post` class.

```
import { Field, Int, ObjectType } from 'type-graphql';

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

  @Field()
  title: string;

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

Since our models are ready, we can move to the resolver class.

```
@Resolver(of => Author)
export class AuthorResolver {
  constructor(
    private readonly authorsService: AuthorsService,
    private readonly postsService: PostsService,
  ) {}

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

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

Conventionally, we would use something like `getAuthor()` or `getPosts()` as method names. We can easily do this by moving the real names to the decorators.

```
@Resolver(of => Author)
export class AuthorResolver {
  constructor(
    private readonly authorsService: AuthorsService,
    private readonly postsService: PostsService,
  ) {}

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

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

Usually, you won't have to pass such an object into the `@Args()` decorator. For example, if your identifier's type would be a string, the following construction would be sufficient:

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

However, the `number` type doesn't give `type-graphql` enough information about the expected GraphQL representation (`Int` vs `Float`) and thus, we have to **explicitly** pass the type reference.

Moreover, you can create a dedicated `AuthorArgs` class:

```
@Args() id: AuthorArgs
```

With the following body:

```
@ArgsType()
class AuthorArgs {
  @Field(type => Int)
  @Min(1)
  id: number;
```

```
}
```

> **HINT**
> Both `@Field()` and `@ArgsType()` decorators are imported from the `type-graphql` package, while `@Min()` comes from the `class-validator` .

You may also notice that such classes play very well with the `ValidationPipe` (read **more**).

Decorators

You may note that we refer to the following arguments using dedicated decorators. Below is a comparison of the provided 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]`

Module

Once we're done here, we have to register the `AuthorResolver` somewhere, for example inside the newly created `AuthorsModule` .

```
@Module({
  imports: [PostsModule],
  providers: [AuthorsService, AuthorResolver],
})
export class AuthorsModule {}
```

The `GraphQLModule` will take care of reflecting the metadata and transforming class into the correct resolvers map automatically. The only thing that you should be aware of is that you need to import this module somewhere, therefore Nest will know that `AuthorsModule` truly exists.

HINT

Learn more about GraphQL queries here.

Support us

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

Principal Sponsor

Sponsors / Partners

Become a sponsor