Tooling

In the GraphQL world, a lot of articles complain how to handle stuff like **authentication**, or **side-effects** of operations. Should we put it inside the business logic? Shall we use a higher-order function to enhance queries and mutations as well, for example, with an authorization logic? Or maybe use **schema directives**. There is no single answer anyway.

Nest ecosystem is trying to help with this issue using existing features like **guards** and **interceptors**. The idea behind them is to reduce redundancy and also, supply you with tooling that helps creating well-structured, readable, and consistent applications.

Overview

You can use either **guards**, **interceptors**, **filters** or **pipes** in the same fashion as in the simple REST application. Additionally, you are able to easily create your own decorators, by leveraging **custom decorators** feature. They all act equivalently. Let's have a look at the following code:

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

As you can see, GraphQL works pretty well with both guards and pipes. Thanks to that you can, for instance, move your authentication logic to the guard, or even reuse the same guard class as in the REST application. The interceptors works in the exact same way:

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

Execution context

However, the `ExecutionContext` received by both guards and interceptors is somewhat different. GraphQL resolvers have a separate set of arguments, respectively, `root`, `args`, `context`, and `info`. Hence, we need to transform given `ExecutionContext` to `GqlExecutionContext`, which is basically very simple.

```
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;
  }
}
```

`GqlExecutionContext` exposes corresponding methods for each argument, like `getArgs()`, `getContext()`, and so on. Now we can effortlessly pick up every argument specific for currently processed request.

Exception filters

The **exception filters** are compatible with GraphQL applications as well.

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

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

However, you don't have an access to the native `response` object in this case (as in the HTTP app).

Custom decorators

As mentioned before, the **custom decorators** feature works like a charm with GraphQL resolvers as well. Though, the factory function takes an array of arguments, instead of a `request` object.

```
export const User = createParamDecorator(
  (data, [root, args, ctx, info]) => ctx.user,
);
```

And then:

```
@Mutation()
async upvotePost(
  @User() user: UserEntity,
  @Args('postId') postId: number,
) {}
```

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

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