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,
});
```