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).