Directives

A directive can be attached to a field or fragment inclusion, and can affect execution of the query in any way the server desires (read more here). The GraphQL specification provides several default directives:

- `@include(if: Boolean)` - only include this field in the result if the argument is true
- `@skip(if: Boolean)` - skip this field if the argument is true
- `@deprecated(reason: String)` - marks field as deprecated with message

A directive is an identifier preceded by a `@` character, optionally followed by a list of named arguments, which can appear after almost any element in the GraphQL query and schema languages.

Custom directives

To instruct what should happen when Apollo/Mercurius encounters your directive, you can create a transformer function. This function uses the `mapSchema` function to iterate through locations in your schema (field definitions, type definitions, etc.) and perform corresponding transformations.

```
import { getDirective, MapperKind, mapSchema } from '@graphql-
tools/utils';
import { defaultFieldResolver, GraphQLSchema } from 'graphql';

export function upperDirectiveTransformer(
  schema: GraphQLSchema,
  directiveName: string,
) {
  return mapSchema(schema, {
    [MapperKind.OBJECT_FIELD]: (fieldConfig) => {
      const upperDirective = getDirective(
        schema,
        fieldConfig,
        directiveName,
      )?.[0];

      if (upperDirective) {
        const { resolve = defaultFieldResolver } = fieldConfig;

        // Replace the original resolver with a function that *first*
calls
        // the original resolver, then converts its result to upper case
        fieldConfig.resolve = async function (source, args, context, info)
{
          const result = await resolve(source, args, context, info);
          if (typeof result === 'string') {
            return result.toUpperCase();
          }
          return result;
        };
        return fieldConfig;
      }
    },
  });
}
```

Now, apply the `upperDirectiveTransformer` transformation function in the `GraphQLModule#forRoot` method using the `transformSchema` function:

```
GraphQLModule.forRoot({
  // ...
  transformSchema: (schema) => upperDirectiveTransformer(schema, 'upper'),
});
```

Once registered, the `@upper` directive can be used in our schema. However, the way you apply the directive will vary depending on the approach you use (code first or schema first).

Code first

In the code first approach, use the `@Directive()` decorator to apply the directive.

```
@Directive('@upper')
@Field()
title: string;
```

> info **Hint** The `@Directive()` decorator is exported from the `@nestjs/graphql` package.

Directives can be applied on fields, field resolvers, input and object types, as well as queries, mutations, and subscriptions. Here's an example of the directive applied on the query handler level:

```
@Directive('@deprecated(reason: "This query will be removed in the next
version")')
@Query(returns => Author, { name: 'author' })
async getAuthor(@Args({ name: 'id', type: () => Int }) id: number) {
  return this.authorsService.findOneById(id);
}
```

> warn **Warning** Directives applied through the `@Directive()` decorator will not be reflected in the generated schema definition file.

Lastly, make sure to declare directives in the `GraphQLModule`, as follows:

```
GraphQLModule.forRoot({
  // ...,
  transformSchema: schema => upperDirectiveTransformer(schema, 'upper'),
  buildSchemaOptions: {
    directives: [
      new GraphQLDirective({
        name: 'upper',
        locations: [DirectiveLocation.FIELD_DEFINITION],
      }),
    ],
  },
}),
```

> info **Hint** Both `GraphQLDirective` and `DirectiveLocation` are exported from the `graphql` package.

Schema first

In the schema first approach, apply directives directly in SDL.

```
directive @upper on FIELD_DEFINITION

type Post {
  id: Int!
  title: String! @upper
  votes: Int
}
```