Unions

Union types are very similar to interfaces, but they don't get to specify any common fields between the types (read more here). Unions are useful for returning disjoint data types from a single field.

Code first

To define a GraphQL union type, we must define classes that this union will be composed of. Following the example from the Apollo documentation, we'll create two classes. First, `Book`:

```
import { Field, ObjectType } from '@nestjs/graphql';

@ObjectType()
export class Book {
  @Field()
  title: string;
}
```

And then `Author`:

```
import { Field, ObjectType } from '@nestjs/graphql';

@ObjectType()
export class Author {
  @Field()
  name: string;
}
```

With this in place, register the `ResultUnion` union using the `createUnionType` function exported from the `@nestjs/graphql` package:

```
export const ResultUnion = createUnionType({
  name: 'ResultUnion',
  types: () => [Author, Book] as const,
});
```

> warning **Warning** The array returned by the `types` property of the `createUnionType` function should be given a const assertion. If the const assertion is not given, a wrong declaration file will be generated at compile time, and an error will occur when using it from another project.

Now, we can reference the `ResultUnion` in our query:

```
@Query(returns => [ResultUnion])
search(): Array<typeof ResultUnion> {
```

```
    return [new Author(), new Book()];
  }
```

This will result in generating the following part of the GraphQL schema in SDL:

```
type Author {
  name: String!
}

type Book {
  title: String!
}

union ResultUnion = Author | Book

type Query {
  search: [ResultUnion!]!
}
```

The default `resolveType()` function generated by the library will extract the type based on the value returned from the resolver method. That means returning class instances instead of literal JavaScript object is obligatory.

To provide a customized `resolveType()` function, pass the `resolveType` property to the options object passed into the `createUnionType()` function, as follows:

```
export const ResultUnion = createUnionType({
  name: 'ResultUnion',
  types: () => [Author, Book] as const,
  resolveType(value) {
    if (value.name) {
      return Author;
    }
    if (value.title) {
      return Book;
    }
    return null;
  },
});
```

Schema first

To define a union in the schema first approach, simply create a GraphQL union with SDL.

```
type Author {
  name: String!
```

```
}

type Book {
  title: String!
}

union ResultUnion = Author | Book
```

Then, you can use the typings generation feature (as shown in the quick start chapter) to generate corresponding TypeScript definitions:

```
export class Author {
  name: string;
}

export class Book {
  title: string;
}

export type ResultUnion = Author | Book;
```

Unions require an extra `__resolveType` field in the resolver map to determine which type the union should resolve to. Also, note that the `ResultUnionResolver` class has to be registered as a provider in any module. Let's create a `ResultUnionResolver` class and define the `__resolveType` method.

```
@Resolver('ResultUnion')
export class ResultUnionResolver {
  @ResolveField()
  __resolveType(value) {
    if (value.name) {
      return 'Author';
    }
    if (value.title) {
      return 'Book';
    }
    return null;
  }
}
```

> info **Hint** All decorators are exported from the `@nestjs/graphql` package.

Enums

Enumeration types are a special kind of scalar that is restricted to a particular set of allowed values (read more here). This allows you to:

- validate that any arguments of this type are one of the allowed values

- communicate through the type system that a field will always be one of a finite set of values

Code first

When using the code first approach, you define a GraphQL enum type by simply creating a TypeScript enum.

```
export enum AllowedColor {
  RED,
  GREEN,
  BLUE,
}
```

With this in place, register the `AllowedColor` enum using the `registerEnumType` function exported from the `@nestjs/graphql` package:

```
registerEnumType(AllowedColor, {
  name: 'AllowedColor',
});
```

Now you can reference the `AllowedColor` in our types:

```
@Field(type => AllowedColor)
favoriteColor: AllowedColor;
```

This will result in generating the following part of the GraphQL schema in SDL:

```
enum AllowedColor {
  RED
  GREEN
  BLUE
}
```

To provide a description for the enum, pass the `description` property into the `registerEnumType()` function.

```
registerEnumType(AllowedColor, {
  name: 'AllowedColor',
  description: 'The supported colors.',
});
```

To provide a description for the enum values, or to mark a value as deprecated, pass the `valuesMap` property, as follows:

```
registerEnumType(AllowedColor, {
  name: 'AllowedColor',
  description: 'The supported colors.',
  valuesMap: {
    RED: {
      description: 'The default color.',
    },
    BLUE: {
      deprecationReason: 'Too blue.',
    },
  },
});
```

This will generate the following GraphQL schema in SDL:

```
"""
The supported colors.
"""
enum AllowedColor {
  """
  The default color.
  """
  RED
  GREEN
  BLUE @deprecated(reason: "Too blue.")
}
```

Schema first

To define an enumerator in the schema first approach, simply create a GraphQL enum with SDL.

```
enum AllowedColor {
  RED
  GREEN
  BLUE
}
```

Then you can use the typings generation feature (as shown in the quick start chapter) to generate corresponding TypeScript definitions:

```
export enum AllowedColor {
  RED
```

```
  GREEN
  BLUE
}
```

Sometimes a backend forces a different value for an enum internally than in the public API. In this example the API contains `RED`, however in resolvers we may use `#f00` instead (read more here). To accomplish this, declare a resolver object for the `AllowedColor` enum:

```
export const allowedColorResolver: Record<keyof typeof AllowedColor, any>
= {
  RED: '#f00',
};
```

> info **Hint** All decorators are exported from the `@nestjs/graphql` package.

Then use this resolver object together with the `resolvers` property of the `GraphQLModule#forRoot()` method, as follows:

```
GraphQLModule.forRoot({
  resolvers: {
    AllowedColor: allowedColorResolver,
  },
});
```