Mutations

Most discussions of GraphQL focus on data fetching, but any complete data platform needs a way to modify server-side data as well. In REST, any request could end up causing side-effects on the server, but best practice suggests we should not modify data in GET requests. GraphQL is similar - technically any query could be implemented to cause a data write. However, like REST, it's recommended to observe the convention that any operations that cause writes should be sent explicitly via a mutation (read more here).

The official Apollo documentation uses an `upvotePost()` mutation example. This mutation implements a method to increase a post's `votes` property value. To create an equivalent mutation in Nest, we'll make use of the `@Mutation()` decorator.

Code first

Let's add another method to the `AuthorResolver` used in the previous section (see resolvers).

```
@Mutation(returns => Post)
async upvotePost(@Args({ name: 'postId', type: () => Int }) postId: number) {
  return this.postsService.upvoteById({ id: postId });
}
```

> info **Hint** All decorators (e.g., `@Resolver`, `@ResolveField`, `@Args`, etc.) are exported from the `@nestjs/graphql` package.

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

```
type Mutation {
  upvotePost(postId: Int!): Post
}
```

The `upvotePost()` method takes `postId` (`Int`) as an argument and returns an updated `Post` entity. For the reasons explained in the resolvers section, we have to explicitly set the expected type.

If the mutation needs to take an object as an argument, we can create an **input type**. The input type is a special kind of object type that can be passed in as an argument (read more here). To declare an input type, use the `@InputType()` decorator.

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

@InputType()
export class UpvotePostInput {
  @Field()
  postId: number;
}
```

> info **Hint** The `@InputType()` decorator takes an options object as an argument, so you can, for example, specify the input type's description. Note that, due to TypeScript's metadata reflection system limitations, you must either use the `@Field` decorator to manually indicate a type, or use a CLI plugin.

We can then use this type in the resolver class:

```
@Mutation(returns => Post)
async upvotePost(
  @Args('upvotePostData') upvotePostData: UpvotePostInput,
) {}
```

Schema first

Let's extend our `AuthorResolver` used in the previous section (see resolvers).

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

Note that we assumed above that the business logic has been moved to the `PostsService` (querying the post and incrementing its `votes` property). The logic inside the `PostsService` class can be as simple or sophisticated as needed. The main point of this example is to show how resolvers can interact with other providers.

The last step is to add our mutation to the existing types definition.

```
type Author {
  id: Int!
  firstName: String
  lastName: String
  posts: [Post]
}

type Post {
  id: Int!
  title: String
  votes: Int
}

type Query {
  author(id: Int!): Author
}

type Mutation {
  upvotePost(postId: Int!): Post
}
```

The `upvotePost(postId: Int!): Post` mutation is now available to be called as part of our application's GraphQL API.