**Interfaces**

Interfaces are designed to reduce the number of queries rather than save typing at the server end.

Also, interfaces could be from another location and would be hard to see what they are.

The schema is designed to be a complete source of information which would not be the case if we did not have to specifically list all fields.

All GraphQL operations must specify their selections down to fields that return scalar values (leaf values). For example, they cannot have fields that describe objects without providing further nested selection sets to specify which scalar values to fetch for these objects. The last-nested level of fields should always consist of only fields that describe scalar values. 2.2.2

Graphical user interface, text

Description automatically generated

GraphQL in Action by Samer Buna

Graphical user interface, text, application, email

Description automatically generated

serialize for response

parseValue and parseLiteral for inputs

<https://graphql.org/graphql-js/constructing-types/>

shows how we can make types without GraphQL Schema

Graphical user interface, text, application, email

Description automatically generated

Text

Description automatically generated

Graphical user interface, text, application

Description automatically generated

This query returns all the types this schema supports, and it also includes the descriptions of these types.

\_\_type \_\_typename

\_\_type(name: "Commit")

4.3.1

Root field nullability A general good practice in GraphQL schemas is to make the types of fields non-null, unless you have a reason to distinguish between null and empty. A non-null type can still hold an empty value. For example, a non-null string can be empty, a non-null list can be an empty array, and a non-null object can be one with no properties.

Only use nullable fields if you want to associate an actual semantic meaning with the absence of their values. However, root fields are special because making them nullable has an important consequence.

In GraphQL.js implementations, when an error is thrown in any field’s resolver, the built-in executor resolves that field with null. When an error is thrown in a resolver for a field that is defined as non-null, the executor propagates the nullability to the field’s parent instead. If that parent field is also non-null, the executor continues up the tree until it finds a nullable field.

This means if the root taskMainList field were to be made non-null, then when an error is thrown in its resolver, the nullability will propagate to the Query type (its parent). So the entire data response for a query asking for this field would be null, even if the query had other root fields. This is not ideal. One bad root field should not block the data response of other root fields. When we start implementing this GraphQL API in the next chapter, we will see an example. This is why I made the taskMainList nullable, and it’s why I will make all root fields nullable. The semantic meaning of this nullability is, in this case, “Something went wrong in the resolver of this root field, and we’re allowing it so that a response can still have partial data for other root fields.”

**5.19**

. For example, you can use the GraphQLSchema constructor to create a schema object, the GraphQLObjectType constructor to create an object type, the GraphQLUnionType to create a union type, and many more classes just like these. This format is useful if you need to construct a schema programmatically. It’s more flexible and easier to test, manage, and extend

. For example, you can use the GraphQLSchema constructor to create a schema object, the GraphQLObjectType constructor to create an object type, the GraphQLUnionType to create a union type, and many more classes just like these. This format is useful if you need to construct a schema programmatically. It’s more flexible and easier to test, manage, and extend