**What is a GraphQL fragment?**

what a **fragment** is in GraphQL. A [GraphQL fragment](https://graphql.org/learn/queries/#fragments) is a reusable part of the query. In GraphQL, you may run into situations where you need to query for the same fields in different queries. If you notice that your query has many repetitive fields in multiple areas, you can consolidate them into a reusable unit called a fragment.

A GraphQL fragment lets you build multiple fields, and include them in multiple queries. You can think of it as functions in programming languages, that are reusable units.

*A GraphQL Fragment is a reusable unit of a GraphQL query, which creates a shared piece of query logic.*

**The components of a GraphQL fragment**

Let’s take a look at the different parts of a GraphQL fragment with a sample structure below:

fragment Name on TypeName {

field1

field2

field3

}

A fragment consists of three unique components:

* **Name**: This is the unique name of the fragment (each fragment can have its own name)
* **TypeName**: The type of object the fragment is going to be used on. This indicates which nested object, from the GraphQL schema, this fragment is created on
* **Body**: The last part is the body of the fragment. The body of the fragment defines the fields that will be queried for

**Benefits of using a GraphQL fragment**

So why are fragments cool within a GrapQL query?

* **Reusability** – With fragments, you can organize your queries into reusable units
* **Caching** – [GraphQL clients](https://www.apollographql.com/docs/react/caching/cache-interaction/) make use of fragments, to provide caching options

**Creating GraphQL fragments**

Let’s learn how to create GraphQL fragments with some examples. For the examples in this blog post, I am using [GitHub’s public API](https://developer.github.com/v4/) and writing queries against it. You can follow along by signing into your GitHub account, and executing the queries from the [GitHub GraphQL Explorer](https://developer.github.com/v4/explorer/).

Notice that we are querying for the same fields inside the **owner** field multiple times. This is a good place to create a fragment:

{

googleRepo: repository (owner:"google", name:"WebFundamentals") {

name

owner {

id,

avatarUrl

resourcePath

url

}

}

facebookRepo: repository (owner:"facebook", name:"react") {

name

owner {

id,

avatarUrl

resourcePath

url

}

}

}

We can rewrite our query to use a fragment. Fragments are created with the keyword fragment.

We can create a fragment called ownerInfo. While creating fragments we have to let GraphQL know on which field it is created. In our case, we are creating the fragment on the RepositoryOwner field. Within our fragment definition, we can include all the fields that we are querying for on the RepositoryOwner object. We are adding id, avatarUrl, resourcePath and url as fields to our fragment.

// fragment ownerInfo for RepositoryOwner fields

fragment ownerInfo on RepositoryOwner {

id

avatarUrl

resourcePath

url

}

**Using a GraphQL fragment**

You can then use the fragment that we created in the previous example, within the query by using the … operator and providing the fragment’s name as shown below:

// GraphQL Query with fragments

{

googleRepo: repository(owner: "google", name: "WebFundamentals") {

name

owner {

...ownerInfo //fragment

}

}

facebookRepo: repository(owner: "facebook", name: "react") {

name

owner {

...ownerInfo //fragment

}

}

}

The snippet shown below is the JSON response after using a fragment. Notice, that there won’t be any changes to the response returned with the use of fragments. Fragments simply make your query clean, readable and reusable. It has no effect on the query response that comes back.

// GraphQL JSON Response

{

"data": {

"googleRepo": {

"name": "WebFundamentals",

"owner": {

"id": "MDEyOk9yZ2FuaXphdGlvbjEzNDIwMDQ=",

"avatarUrl": "https://avatars1.githubusercontent.com/u/1342004?v=4",

"resourcePath": "/google",

"url": "https://github.com/google"

}

},

"facebookRepo": {

"name": "react",

"owner": {

"id": "MDEyOk9yZ2FuaXphdGlvbjY5NjMx",

### Enhancing Reusability with Fragments

Fragments are a handy feature to help to improve the structure and reusability of your GraphQL code. A fragment is a collection of fields on a specific type.

Let’s assume we have the following type:

type User {

name: String!

age: Int!

email: String!

street: String!

zipcode: String!

city: String!

}

Here, we could represent all the information that relates to the user’s physical address into a fragment:

fragment addressDetails on User {

name

street

zipcode

city

}

Now, when writing a query to access the address information of a user, we can use the following syntax to refer to the fragment and save the work to actually spell out the four fields:

{

allUsers {

... addressDetails

}

}

This query is equivalent to writing:

{

allUsers {

name

street

zipcode

city

}

}

### Parameterizing Fields with Arguments

In GraphQL type definitions, each field can take zero or more arguments. Similar to arguments that are passed into functions in typed programming languages, each argument needs to have a name and a type. In GraphQL, it’s also possible to specify default values for arguments.

As an example, let’s consider a part of the schema that we saw in the beginning:

type Query {

allUsers: [User!]!

}

type User {

name: String!

age: Int!

}

We could now add an argument to the allUsers field that allows us to pass an argument to filter users and include only those above a certain age. We also specify a default value so that by default all users will be returned:

type Query {

allUsers(olderThan: Int = -1): [User!]!

}

This olderThan argument can now be passed into the query using the following syntax:

{

allUsers(olderThan: 30) {

name

age

}

}

### Named Query Results with Aliases

One of GraphQL’s major strengths is that it lets you send multiple queries in a single request. However, since the response data is shaped after the structure of the fields being requested, you might run into naming issues when you’re sending multiple queries asking for the same fields:

{

User(id: "1") {

name

}

User(id: "2") {

name

}

}

In fact, this will produce an error with a GraphQL server, since it’s the same field but different arguments. The only way to send a query like that would be to use aliases, i.e. specifying names for the query results:

{

first: User(id: "1") {

name

}

second: User(id: "2") {

name

}

}

In the result, the server would now name each User object according to the specified alias:

{

"first": {

"name": "Alice"

},

"second": {

"name": "Sarah"

}

}

### Advanced SDL

The SDL offers a couple of language features that weren’t discussed in the previous chapter. In the following, we’ll discuss those by practical examples.

#### Object & Scalar Types

In GraphQL, there are two different kinds of types.

* Scalar types represent concrete units of data. The GraphQL spec has five predefined scalars: as String, Int, Float, Boolean, and ID.
* Object types have fields that express the properties of that type and are composable. Examples of object types are the User or Post types we saw in the previous section.

In every GraphQL schema, you can define your own scalar and object types. An often cited example for a custom scalar would be a Date type where the implementation needs to define how that type is validated, serialized, and deserialized.

#### Enums

GraphQL allows you to define enumerations types (short enums), a language feature to express the semantics of a type that has a fixed set of values. We could thus define a type called Weekday to represent all the days of a week:

enum Weekday {

MONDAY

TUESDAY

WEDNESDAY

THURSDAY

FRIDAY

SATURDAY

SUNDAY

}

Note that technically enums are special kinds of scalar types.

#### Interface

An interface can be used to describe a type in an abstract way. It allows you to specify a set of fields that any concrete type, which implements this interface, needs to have. In many GraphQL schemas, every type is required to have an id field. Using interfaces, this requirement can be expressed by defining an interface with this field and then making sure that all custom types implement it:

interface Node {

id: ID!

}

type User implements Node {

id: ID!

name: String!

age: Int!

}

#### Union Types

Union types can be used to express that a type should be either of a collection of other types. They are best understood by means of an example. Let’s consider the following types:

type Adult {

name: String!

work: String!

}

type Child {

name: String!

school: String!

}

Now, we could define a Person type to be the union of Adult and Child:

union Person = Adult | Child

This brings up a different problem: In a GraphQL query where we ask to retrieve information about a Child but only have a Person type to work with, how do we know whether we can actually access this field?

The answer to this is called conditional fragments:

{

allPersons {

name *# works for `Adult` and `Child`*

... on Child {

school

}

... on Adult {

work

}

}

}