Queries and Mutations

On this page, you'll learn in detail about how to query a GraphQL server.

Fields

At its simplest, GraphQL is about asking for specific fields on objects. Let's start by looking at a very simple query and the result we get when we run it:

```
{
  hero {
    name
  }
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2"
    }
  }
}
```

You can see immediately that the query has exactly the same shape as the result. This is essential to GraphQL, because you always get back what you expect, and the server knows exactly what fields the client is asking for.

The field `name` returns a `String` type, in this case the name of the main hero of Star Wars, `"R2-D2"`.

> *Oh, one more thing - the query above is interactive. That means you can change it as you like and see the new result. Try adding an* `appearsIn` *field to the* `hero` *object in the query, and see the new result.*

In the previous example, we just asked for the name of our hero which returned a String, but fields can also refer to Objects. In that case, you can make a *sub-selection* of fields for that object. GraphQL queries can traverse related objects

and their fields, letting clients fetch lots of related data in one request, instead of making several roundtrips as one would need in a classic REST architecture.

```
{
  hero {
    name
    # Queries can have comments!
    friends {
      name
    }
  }
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friends": [
        {
          "name": "Luke Skywalke
        },
        {
          "name": "Han Solo"
        },
        {
```

Note that in this example, the `friends` field returns an array of items. GraphQL queries look the same for both single items or lists of items; however, we know which one to expect based on what is indicated in the schema.

Arguments

If the only thing we could do was traverse objects and their fields, GraphQL would already be a very useful language for data fetching. But when you add the ability to pass arguments to fields, things get much more interesting.

```
{
  human(id: "1000") {
    name
    height
  }
}
```

```
{
  "data": {
    "human": {
      "name": "Luke Skywalker",
      "height": 1.72
    }
  }
}
```

In a system like REST, you can only pass a single set of arguments - the query parameters and URL segments in your request. But in GraphQL, every field and nested object can get its own set of arguments, making GraphQL a complete replacement for making multiple API fetches. You can even pass arguments into scalar fields, to implement data transformations once on the server, instead of on every client separately.

```
{
  human(id: "1000") {
    name
    height(unit: FOOT)
  }
}
```

```
{
  "data": {
    "human": {
      "name": "Luke Skywalker",
      "height": 5.6430448
    }
  }
}
```

Arguments can be of many different types. In the above example, we have used an Enumeration type, which represents one of a finite set of options (in this case, units of length, either `METER` or `FOOT`). GraphQL comes with a default set of types, but a GraphQL server can also declare its own custom types, as long as they can be serialized into your transport format.

Read more about the GraphQL type system here.

Aliases

If you have a sharp eye, you may have noticed that, since the result object fields match the name of the field in the query but don't include arguments, you can't directly query for the same field with different arguments. That's why you need *aliases* - they let you rename the result of a field to anything you want.

```
{
  empireHero: hero(episode: EMPI
    name
  }
  jediHero: hero(episode: JEDI)
    name
  }
}
```

```
{
  "data": {
    "empireHero": {
      "name": "Luke Skywalker"
    },
    "jediHero": {
      "name": "R2-D2"
    }
  }
}
```

In the above example, the two `hero` fields would have conflicted, but since we can alias them to different names, we can get both results in one request.

Fragments

Let's say we had a relatively complicated page in our app, which lets us look at two heroes side by side, along with their friends. You can imagine that such a query could quickly get complicated, because we would need to repeat the fields at least once - one for each side of the comparison.

That's why GraphQL includes reusable units called *fragments*. Fragments let you construct sets of fields, and then include them in queries where you need to. Here's an example of how you could solve the above situation using fragments:

```
{
  leftComparison: hero(episode: I
    ...comparisonFields
  }
  rightComparison: hero(episode:
    ...comparisonFields
  }
}

fragment comparisonFields on Cha
  name
  appearsIn
  friends {
    name
  }
}
```

```
{
  "data": {
    "leftComparison": {
      "name": "Luke Skywalker",
      "appearsIn": [
        "NEWHOPE",
        "EMPIRE",
        "JEDI"
      ],
      "friends": [
        {
          "name": "Han Solo"
        },
        {
          "name": "Leia Organa"
        },
        {
          "name": "C-3PO"
        },
        {
```

You can see how the above query would be pretty repetitive if the fields were repeated. The concept of fragments is frequently used to split complicated application data requirements into smaller chunks, especially when you need to combine lots of UI components with different fragments into one initial data fetch.

Using variables inside fragments

It is possible for fragments to access variables declared in the query or mutation. See variables.

```
query HeroComparison($first: Int
  leftComparison: hero(episode: I
    ...comparisonFields
  }
  rightComparison: hero(episode:
    ...comparisonFields
  }
}

fragment comparisonFields on Cha
  name
  friendsConnection(first: $firs
    totalCount
    edges {
      node {
        name
      }
    }
  }
}
```

```
{
  "data": {
    "leftComparison": {
      "name": "Luke Skywalker",
      "friendsConnection": {
        "totalCount": 4,
        "edges": [
          {
            "node": {
              "name": "Han Solo"
            }
          },
          {
            "node": {
              "name": "Leia Orga
            }
          },
          {
            "node": {
              "name": "C-3PO"
            }
          }
        ]
      }
    },
    "rightComparison": {
```

Operation name

In several of the examples above we have been using a shorthand syntax where we omit both the `query` keyword and the query name, but in production apps it's useful to use these to make our code less ambiguous.

Here's an example that includes the keyword `query` as *operation type* and `HeroNameAndFriends` as *operation name*:

```
query HeroNameAndFriends {
  hero {
    name
    friends {
      name
    }
  }
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friends": [
        {
          "name": "Luke Skywalke
        },
```

```
}
```

```
{
  "name": "Han Solo"
},
{
```

The *operation type* is either *query*, *mutation*, or *subscription* and describes what type of operation you're intending to do. The operation type is required unless you're using the query shorthand syntax, in which case you can't supply a name or variable definitions for your operation.

The *operation name* is a meaningful and explicit name for your operation. It is only required in multi-operation documents, but its use is encouraged because it is very helpful for debugging and server-side logging. When something goes wrong (you see errors either in your network logs, or in the logs of your GraphQL server) it is easier to identify a query in your codebase by name instead of trying to decipher the contents. Think of this just like a function name in your favorite programming language. For example, in JavaScript we can easily work only with anonymous functions, but when we give a function a name, it's easier to track it down, debug our code, and log when it's called. In the same way, GraphQL query and mutation names, along with fragment names, can be a useful debugging tool on the server side to identify different GraphQL requests.

Variables

So far, we have been writing all of our arguments inside the query string. But in most applications, the arguments to fields will be dynamic: For example, there might be a dropdown that lets you select which Star Wars episode you are interested in, or a search field, or a set of filters.

It wouldn't be a good idea to pass these dynamic arguments directly in the query string, because then our client-side code would need to dynamically manipulate the query string at runtime, and serialize it into a GraphQL-specific format. Instead, GraphQL has a first-class way to factor dynamic values out of the query, and pass them as a separate dictionary. These values are called *variables*.

When we start working with variables, we need to do three things:

1. Replace the static value in the query with `$variableName`
2. Declare `$variableName` as one of the variables accepted by the query
3. Pass `variableName: value` in the separate, transport-specific (usually JSON) variables dictionary

Here's what it looks like all together:

```
query HeroNameAndFriends($episod
  hero(episode: $episode) {
    name
    friends {
      name
    }
  }
}
```

```
{
  "episode": "JEDI"
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friends": [
        {
          "name": "Luke Skywalke
        },
        {
          "name": "Han Solo"
        },
        {
          "name": "Leia Organa"
        }
      ]
    }
```

Now, in our client code, we can simply pass a different variable rather than needing to construct an entirely new query. This is also in general a good practice for denoting which arguments in our query are expected to be dynamic - we should never be doing string interpolation to construct queries from user-supplied values.

Variable definitions

The variable definitions are the part that looks like `($episode: Episode)` in the query above. It works just like the argument definitions for a function in a typed language. It lists all of the variables, prefixed by `$`, followed by their type, in this case `Episode`.

All declared variables must be either scalars, enums, or input object types. So if you want to pass a complex object into a field, you need to know what input

type that matches on the server. Learn more about input object types on the Schema page.

Variable definitions can be optional or required. In the case above, since there isn't an `!` next to the `Episode` type, it's optional. But if the field you are passing the variable into requires a non-null argument, then the variable has to be required as well.

To learn more about the syntax for these variable definitions, it's useful to learn the GraphQL schema language. The schema language is explained in detail on the Schemas and Types page.

Default variables

Default values can also be assigned to the variables in the query by adding the default value after the type declaration.

```
query HeroNameAndFriends($episode: Episode = JEDI) {
  hero(episode: $episode) {
    name
    friends {
      name
    }
  }
}
```

When default values are provided for all variables, you can call the query without passing any variables. If any variables are passed as part of the variables dictionary, they will override the defaults.

Directives

We discussed above how variables enable us to avoid doing manual string interpolation to construct dynamic queries. Passing variables in arguments solves a pretty big class of these problems, but we might also need a way to dynamically change the structure and shape of our queries using variables. For

example, we can imagine a UI component that has a summarized and detailed view, where one includes more fields than the other.

Let's construct a query for such a component:

```
query Hero($episode: Episode, $w
  hero(episode: $episode) {
    name
    friends @include(if: $withFr
      name
    }
  }
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2"
    }
  }
}
```

```
{
  "episode": "JEDI",
  "withFriends": false
}
```

Try editing the variables above to instead pass `true` for `withFriends`, and see how the result changes.

We needed to use a new feature in GraphQL called a *directive*. A directive can be attached to a field or fragment inclusion, and can affect execution of the query in any way the server desires. The core GraphQL specification includes exactly two directives, which must be supported by any spec-compliant GraphQL server implementation:

- `@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`.

Directives can be useful to get out of situations where you otherwise would need to do string manipulation to add and remove fields in your query. Server implementations may also add experimental features by defining completely new directives.

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 might end up causing some side-effects on the server, but by convention it's suggested that one doesn't use `GET` requests to modify data. GraphQL is similar - technically any query could be implemented to cause a data write. However, it's useful to establish a convention that any operations that cause writes should be sent explicitly via a mutation.

Just like in queries, if the mutation field returns an object type, you can ask for nested fields. This can be useful for fetching the new state of an object after an update. Let's look at a simple example mutation:

```
mutation CreateReviewForEpisode(
  createReview(episode: $ep, rev
    stars
    commentary
  }
}
```

```
{
  "ep": "JEDI",
  "review": {
    "stars": 5,
    "commentary": "This is a gre
  }
}
```

```
{
  "data": {
    "createReview": {
      "stars": 5,
      "commentary": "This is a g
    }
  }
}
```

Note how `createReview` field returns the `stars` and `commentary` fields of the newly created review. This is especially useful when mutating existing data, for example, when incrementing a field, since we can mutate and query the new value of the field with one request.

You might also notice that, in this example, the `review` variable we passed in is not a scalar. It's an *input object type*, a special kind of object type that can be

passed in as an argument. Learn more about input types on the Schema page.

Multiple fields in mutations

A mutation can contain multiple fields, just like a query. There's one important distinction between queries and mutations, other than the name:

While query fields are executed in parallel, mutation fields run in series, one after the other.

This means that if we send two `incrementCredits` mutations in one request, the first is guaranteed to finish before the second begins, ensuring that we don't end up with a race condition with ourselves.

Inline Fragments

Like many other type systems, GraphQL schemas include the ability to define interfaces and union types. Learn about them in the schema guide.

If you are querying a field that returns an interface or a union type, you will need to use *inline fragments* to access data on the underlying concrete type. It's easiest to see with an example:

```
query HeroForEpisode($ep: Episod
  hero(episode: $ep) {
    name
    ... on Droid {
      primaryFunction
    }
    ... on Human {
      height
    }
  }
}
```

```
{
  "ep": "JEDI"
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "primaryFunction": "Astrom
    }
  }
}
```

```
}
```

In this query, the `hero` field returns the type `Character`, which might be either a `Human` or a `Droid` depending on the `episode` argument. In the direct selection, you can only ask for fields that exist on the `Character` interface, such as `name`.

To ask for a field on the concrete type, you need to use an *inline fragment* with a type condition. Because the first fragment is labeled as `... on Droid`, the `primaryFunction` field will only be executed if the `Character` returned from `hero` is of the `Droid` type. Similarly for the `height` field for the `Human` type.

Named fragments can also be used in the same way, since a named fragment always has a type attached.

Meta fields

Given that there are some situations where you don't know what type you'll get back from the GraphQL service, you need some way to determine how to handle that data on the client. GraphQL allows you to request `__typename`, a meta field, at any point in a query to get the name of the object type at that point.

```
{
  search(text: "an") {
    __typename
    ... on Human {
      name
    }
    ... on Droid {
      name
    }
    ... on Starship {
      name
    }
  }
}
```

```
{
  "data": {
    "search": [
      {
        "__typename": "Human",
        "name": "Han Solo"
      },
      {
        "__typename": "Human",
        "name": "Leia Organa"
      },
      {
        "__typename": "Starship"
        "name": "TIE Advanced x1
      }
    ]
  }
```

In the above query, `search` returns a union type that can be one of three options. It would be impossible to tell apart the different types from the client without the `__typename` field.

GraphQL services provide a few meta fields, the rest of which are used to expose the Introspection system.

Last updated on May 23, 2024

‹ Introduction

Schemas and Types ›