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, $withFrie
  hero(episode: $episode) {
    name
    friends @include(if: $withFriends)
      name
    }
  }
}


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

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

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($ep: Ep
  createReview(episode: $ep, review: $r
    stars
    commentary
  }
}


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

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

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: Episode!) {
  hero(episode: $ep) {
    name
    ... on Droid {
      primaryFunction
    }
    ... on Human {
      height
    }
  }
}

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

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

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.