

GraphQL Execution

GraphQL ¹⁾ JSON

GraphQL ²⁾

```
type Query {
  human(id: ID!): Human
}

type Human {
  name: String
  appearsIn: [Episode]
  starships: [Starship]
}

enum Episode {
  NEWHOPE
  EMPIRE
  JEDI
}

type Starship {
  name: String
}
```

가 ³⁾

```
{
  human(id: 1002) {
    name
    appearsIn
    starships {
      name
    }
  }
}

{
  "data": {
    "human": {
      "name": "Han Solo",
      "appearsIn": [
        "NEWHOPE",
        "EMPIRE",
        "JEDI"
      ],
      "starships": [
        {
          "name": "Millenium
Falcon"
        },
        {
          "name": "Imperial
```

```
shuttle"
  }
]
}
}
```

GraphQL

GraphQL

가

GraphQL

가

4)

가

. GraphQL

5)

가

Root fields & resolvers

GraphQL

GraphQL API

가

6)

id

human

Human

7)

```
Query: {
  human(obj, args, context, info) {
    return context.db.loadHumanByID(args.id).then(
      userData => new Human(userData)
    )
  }
}
```

JavaScript

GraphQL

4

8)

- obj:
- args: GraphQL
- context:

9)

10)

11)

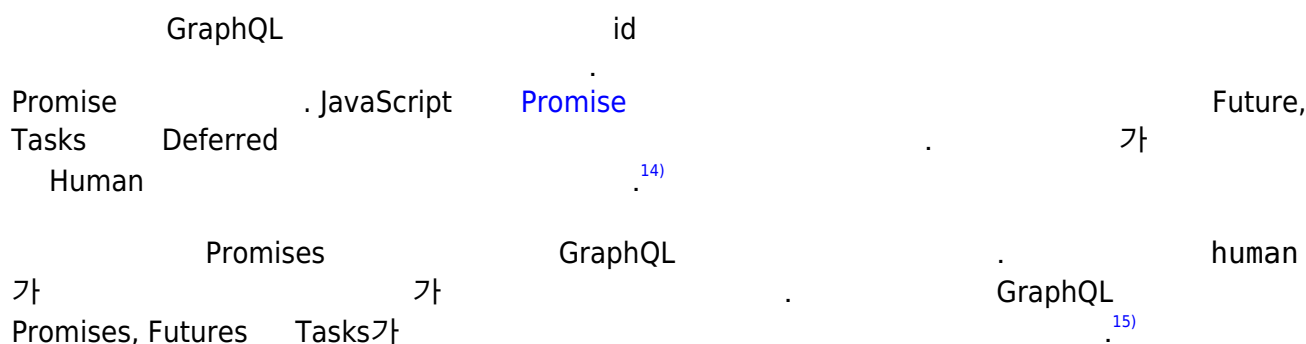
- info:
GraphQLResolveInfo

12)

Asynchronous resolvers

13)

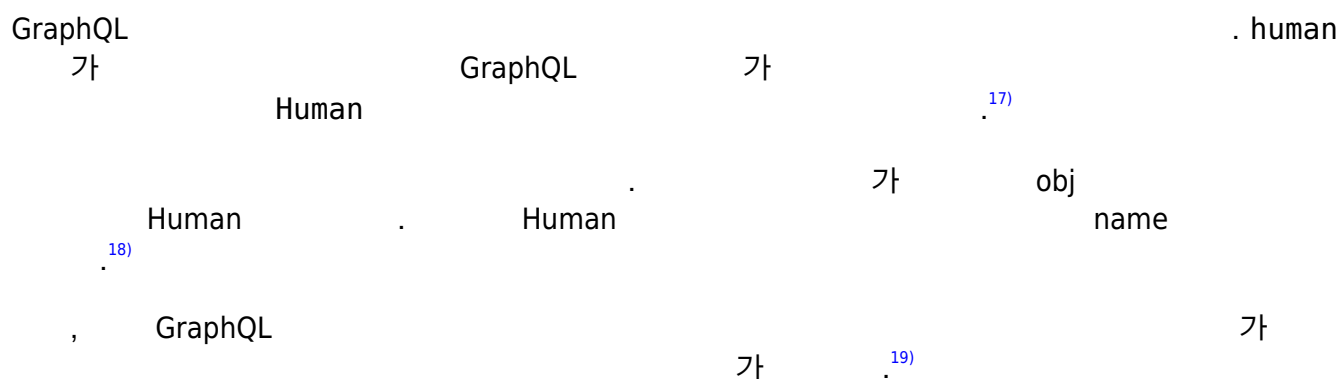
```
human(obj, args, context, info) {
    return context.db.loadHumanByID(args.id).then(
        userData => new Human(userData)
    )
}
```



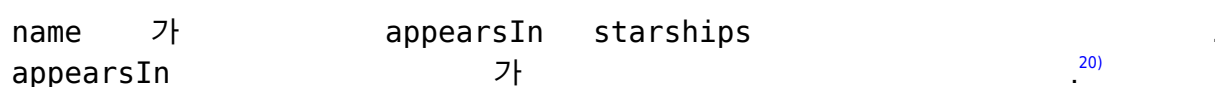
Trivial resolvers



```
Human: {
  name(obj, args, context, info) {
    return obj.name
  }
}
```

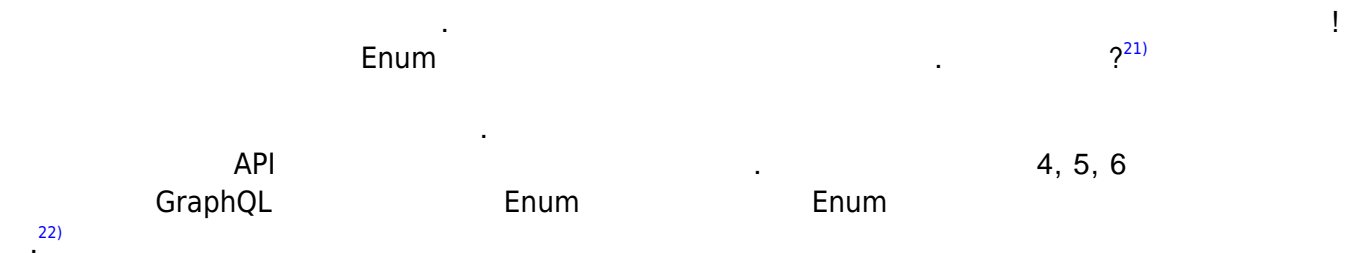


Scalar coercion

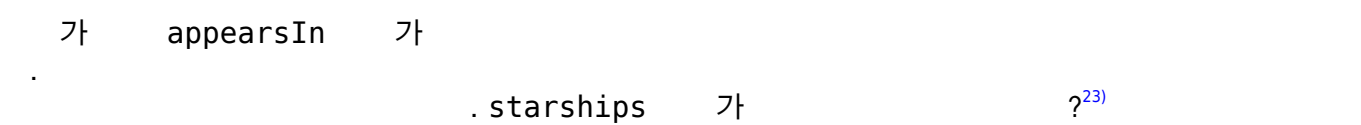


```
Human: {
  appearsIn(obj) {
```

```
return obj.appearsIn // returns [ 4, 5, 6 ]
}
```



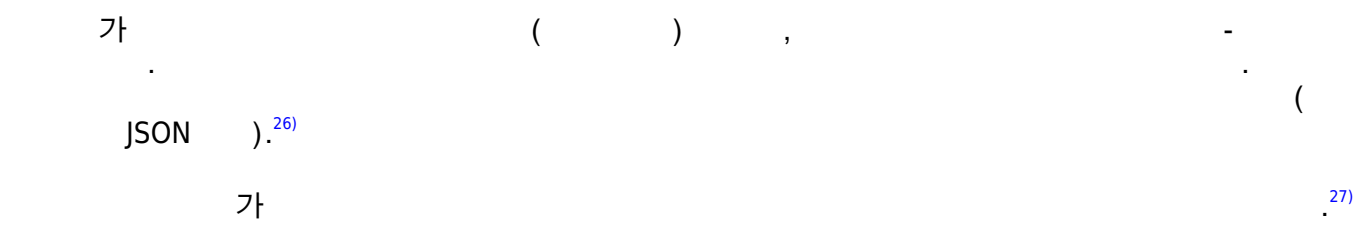
List resolvers



```
Human: {
  starships(obj, args, context, info) {
    return obj.starshipIDs.map(
      id => context.db.loadStarshipByID(id).then(
        shipData => new Starship(shipData)
      )
    )
  }
}
```



Producing the result



```

{
  human(id: 1002) {
    name
    appearsIn
    starships {
      name
    }
  }
}

{
  "data": {
    "human": {
      "name": "Han Solo",
      "appearsIn": [
        "NEWHOPE",
        "EMPIRE",
        "JEDI"
      ],
      "starships": [
        {
          "name": "Millenium
Falcon"
        },
        {
          "name": "Imperial
shuttle"
        }
      ]
    }
  }
}

```

Continue Reading

- [Introspection](#)

Plugin Backlinks:

1)

After being validated, a GraphQL query is executed by a GraphQL server which returns a result that mirrors the shape of the requested query, typically as JSON.

2)

GraphQL cannot execute a query without a type system, let's use an example type system to illustrate executing a query. This is a part of the same type system used throughout the examples in these articles:

3)

In order to describe what happens when a query is executed, let's use an example to walk through.

4)

You can think of each field in a GraphQL query as a function or method of the previous type which returns the next type. In fact, this is exactly how GraphQL works. Each field on each type is backed by a function called the resolver which is provided by the GraphQL server developer. When a field is executed, the corresponding resolver is called to produce the next value.

5)

If a field produces a scalar value like a string or number, then the execution completes. However if a field produces an object value then the query will contain another selection of fields which apply to that object. This continues until scalar values are reached. GraphQL queries always end at scalar

values.

6)

At the top level of every GraphQL server is a type that represents all of the possible entry points into the GraphQL API, it's often called the Root type or the Query type.

7)

In this example, our Query type provides a field called human which accepts the argument id. The resolver function for this field likely accesses a database and then constructs and returns a Human object.

8)

This example is written in JavaScript, however GraphQL servers can be built in many different languages. A resolver function receives four arguments:

9)

The previous object, which for a field on the root Query type is often not used.

10)

The arguments provided to the field in the GraphQL query.

11)

12)

A value which holds field-specific information relevant to the current query as well as the schema details, also refer to type [GraphQLResolveInfo](#) for more details.

13)

Let's take a closer look at what's happening in this resolver function.

14)

The context is used to provide access to a database which is used to load the data for a user by the id provided as an argument in the GraphQL query. Since loading from a database is an asynchronous operation, this returns a Promise. In JavaScript, Promises are used to work with asynchronous values, but the same concept exists in many languages, often called Futures, Tasks or Deferred. When the database returns, we can construct and return a new Human object.

15)

Notice that while the resolver function needs to be aware of Promises, the GraphQL query does not. It simply expects the human field to return something which it can then ask the name of. During execution, GraphQL will wait for Promises, Futures, and Tasks to complete before continuing and will do so with optimal concurrency.

16)

Now that a Human object is available, GraphQL execution can continue with the fields requested on it.

17)

A GraphQL server is powered by a type system which is used to determine what to do next. Even before the human field returns anything, GraphQL knows that the next step will be to resolve fields on the Human type since the type system tells it that the human field will return a Human.

18)

Resolving the name in this case is very straight-forward. The name resolver function is called and the obj argument is the new Human object returned from the previous field. In this case, we expect that Human object to have a name property which we can read and return directly.

19)

In fact, many GraphQL libraries will let you omit resolvers this simple and will just assume that if a resolver isn't provided for a field, that a property of the same name should be read and returned.

20)

While the name field is being resolved, the appearsIn and starships fields can be resolved concurrently. The appearsIn field could also have a trivial resolver, but let's take a closer look:

21)

Notice that our type system claims appearsIn will return Enum values with known values, however this function is returning numbers! Indeed if we look up at the result we'll see that the appropriate Enum values are being returned. What's going on?

22)

This is an example of scalar coercion. The type system knows what to expect and will convert the values returned by a resolver function into something that upholds the API contract. In this case, there may be an Enum defined on our server which uses numbers like 4, 5, and 6 internally, but represents them as Enum values in the GraphQL type system.

23)

We've already seen a bit of what happens when a field returns a list of things with the `appearsIn` field above. It returned a list of enum values, and since that's what the type system expected, each item in the list was coerced to the appropriate enum value. What happens when the `starships` field is resolved?

24)

The resolver for this field is not just returning a Promise, it's returning a list of Promises. The `Human` object had a list of ids of the Starships they piloted, but we need to go load all of those ids to get real Starship objects.

25)

GraphQL will wait for all of these Promises concurrently before continuing, and when left with a list of objects, it will concurrently continue yet again to load the `name` field on each of these items.

26)

As each field is resolved, the resulting value is placed into a key-value map with the field name (or alias) as the key and the resolved value as the value. This continues from the bottom leaf fields of the query all the way back up to the original field on the root `Query` type. Collectively these produce a structure that mirrors the original query which can then be sent (typically as JSON) to the client which requested it.

27)

Let's take one last look at the original query to see how all these resolving functions produce a result:

From:

<https://moro.kr/> - **Various Ways**

Permanent link:

<https://moro.kr/open/graphql-execution>

Last update: **2022/09/01 08:28**

