

Schemas and Types

- <https://graphql.org/learn/schema/>

GraphQL

. GraphQL

1)

Type System

GraphQL

GraphQL

가

2)

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

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "appearsIn": [
        "NEWHOPE",
        "EMPIRE",
        "JEDI"
      ]
    }
  }
}
```

```
1.    root
```

3)

2. hero

4)

3. hero

name

```
appearsIn
```

5

GraphQL

가

가

?

?

가

6)

?

GraphQL

가

가

7)

Type Language

GraphQL

JavaScript

. GraphQL

. “GraphQL

”

GraphQL

8)

Object Types and Fields

GraphQL 가 . GraphQL 9)

```
type Character {
  name: String!
  appearsIn: [Episode!]!
}
```

가 10)

- Character GraphQL Object Type , 가 . 11)
- name appearsIn Character . name appearsIn Character GraphQL 12)
- String 가 . 13)
- String! 가 non-nullable , GraphQL . type language 14)
- [Episode!]! Episode non-nullable appearsIn Episode! non-nullable (0) 15)

GraphQL , GraphQL

Arguments

GraphQL 0 가 (, length) 16)

```
type Starship {
  id: ID!
  name: String!
  length(unit: LengthUnit = METER): Float
}
```

GraphQL JavaScript Python length 17)

The Query and Mutation Types

가 .

```
schema {
  query: Query
  mutation: Mutation
}
```

GraphQL query , mutation GraphQL .
가

```
query {
  hero {
    name
  }
  droid(id: "2000") {
    name
  }
}

{
  "data": {
    "hero": {
      "name": "R2-D2"
    },
    "droid": {
      "name": "C-3PO"
    }
  }
}
```

, GraphQL hero droid 가 Query ¹⁸⁾

```
type Query {
  hero(episode: Episode): Character
  droid(id: ID!): Droid
}
```

Mutations . Mutation ¹⁹⁾
root mutation .

“ ” Query Mutation GraphQL ²⁰⁾

Scala Types

GraphQL 가 ²¹⁾

name appearsIn .

```
{
  hero {
    name
    appearsIn
  }
}

{
  "data": {
    "hero": {
      "name": "R2-D2",
      "appearsIn": [
        "NEWHOPE",
        "EMPIRE",
        "JEDI"
      ]
    }
  }
}
```

가 . 22)

GraphQL 23)

- Int: 32 24)
- Float: 25)
- String: UTF-8 26)
- Boolean: true false. 27)
- ID: ID 가 ID 28)

GraphQL , 29)

Date

```
scalar Date
```

Date , 30)

Enumeration Types

Enums 31)

1. 가 32)
2. 가 33)

GraphQL 34)

```
enum Episode {
  NEWHOPE
}
```

```
EMPIRE
JEDI
}
```

Episode

NEWHOPE, EMPIRE

JEDI

GraphQL

JavaScript

Lists and Non-Null

GraphQL

가

```
type Character {
  name: String!
  appearsIn: [Episode]!
}
```

String

가

! 가 Non-Null

GraphQL

GraphQL

가

GraphQL

```
query DroidById($id: ID!) {
  droid(id: $id) {
    name
  }
}
```

```
VARIABLES
{
  "id": null
}
```

```
{
  "errors": [
    {
      "message": "Variable \"$id\"
of non-null type \"ID!\" must not
be null.",
      "locations": [
        {
          "line": 1,
          "column": 17
        }
      ]
    }
  ]
}
```

]

}

가

[]

40)

.

Non-Null

List

.

Null

41)

.

```
myField: [String!]
```

, null null . JSON :42)

```
myField: null // valid
myField: [] // valid
myField: ['a', 'b'] // valid
myField: ['a', null, 'b'] // error
```

null 가 43)

```
myField: [String]!
```

가 null null 44)

```
myField: null // error
myField: [] // valid
myField: ['a', 'b'] // valid
myField: ['a', null, 'b'] // valid
```

Non-Null List 45)

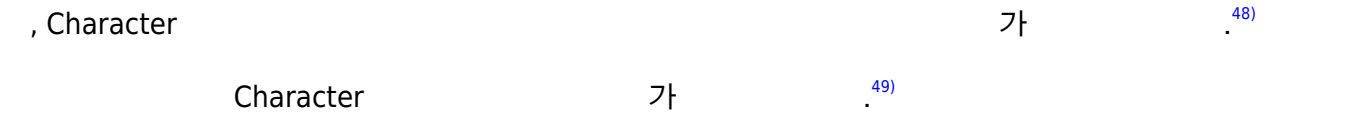
Interfaces

가 GraphQL 46)

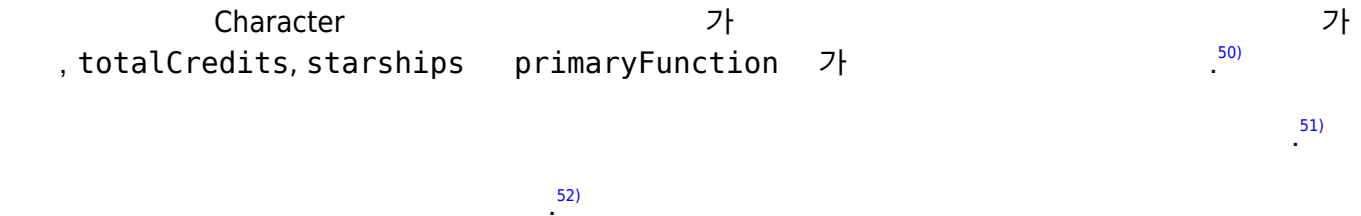
, Star Wars 3 Character 가 47)

```
interface Character {
  id: ID!
```

```
name: String!  
friends: [Character]  
appearsIn: [Episode]!  
}
```



```
type Human implements Character {  
  id: ID!  
  name: String!  
  friends: [Character]  
  appearsIn: [Episode]!  
  starships: [Starship]  
  totalCredits: Int  
}  
  
type Droid implements Character {  
  id: ID!  
  name: String!  
  friends: [Character]  
  appearsIn: [Episode]!  
  primaryFunction: String  
}
```

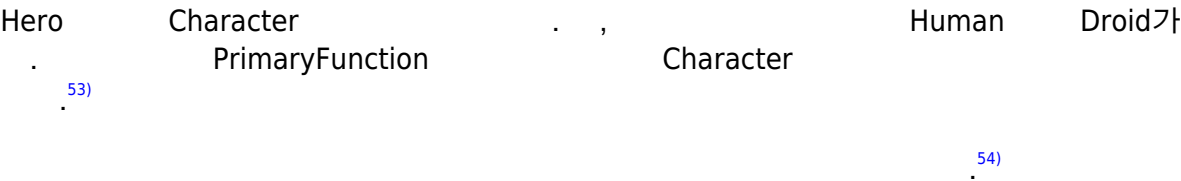


```
query HeroForEpisode($ep: Episode!) {  
  {  
    hero(episode: $ep) {  
      name  
      primaryFunction  
    }  
  }  
  "errors": [  
    {  
      "message": "Cannot query field \"primaryFunction\" on type \"Character\". Did you mean to use an inline fragment on \"Droid\"?",  
      "locations": [  
        {  
          "line": 4,  
          "column": 5  
        }  
      ]  
    }  
  ]  
}
```

VARIABLES

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

```
]
}
```



```
query HeroForEpisode($ep: Episode!) {
  {
    hero(episode: $ep) {
      name
      ... on Droid {
        primaryFunction
      }
    }
  }
  {
    "data": {
      "hero": {
        "name": "R2-D2",
        "primaryFunction":
        "Astromech"
      }
    }
  }
}
```

VARIABLES

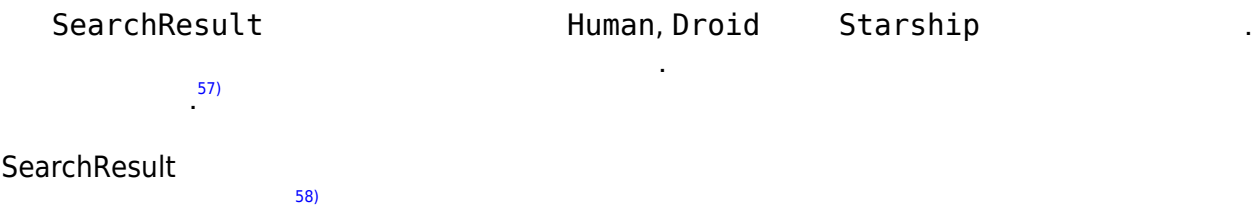
```
{
  "ep": "JEDI"
}
```

가 55)

Union Types

Union 56)

```
union SearchResult = Human | Droid | Starship
```



```
{
  search(text: "an") {
    __typename
    ... on Human {
      {
        "data": {
          "search": [
            {
```



```

    name
    height
}
... on Droid {
    name
    primaryFunction
}
... on Starship {
    name
    length
}
}
}

    "__typename": "Human",
    "name": "Han Solo",
    "height": 1.8
},
{
    "__typename": "Human",
    "name": "Leia Organa",
    "height": 1.5
},
{
    "__typename": "Starship",
    "name": "TIE Advanced x1",
    "length": 9.2
}
]
}
}
}

```

__typename

59)

Human Droid

(Character)

60)

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

Starship

61)

Starship

가

Input Types

mutations

. GraphQL

input

62)

```
input ReviewInput {
  stars: Int!
  commentary: String
}
```

mutation

63)

```
mutation {
  CreateReviewForEpisode($ep: Episode!, $review: ReviewInput!) {
    createReview(episode: $ep, review: $review) {
      stars
      commentary
    }
  }
}
```

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

VARIABLES

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

가

64)

Continue Reading

- [Validation](#)

Plugin Backlinks:

1)

On this page, you'll learn all you need to know about the GraphQL type system and how it describes what data can be queried. Since GraphQL can be used with any backend framework or programming language, we'll stay away from implementation-specific details and talk only about the concepts.

2)

If you've seen a GraphQL query before, you know that the GraphQL query language is basically about selecting fields on objects. So, for example, in the following query:

3)

We start with a special "root" object

4)

We select the hero field on that

5)

For the object returned by hero, we select the name and appearsIn fields

6)

Because the shape of a GraphQL query closely matches the result, you can predict what the query will return without knowing that much about the server. But it's useful to have an exact description of the data we can ask for - what fields can we select? What kinds of objects might they return? What fields are available on those sub-objects? That's where the schema comes in.

7)

Every GraphQL service defines a set of types which completely describe the set of possible data you can query on that service. Then, when queries come in, they are validated and executed against that schema.

8)

GraphQL services can be written in any language. Since we can't rely on a specific programming language syntax, like JavaScript, to talk about GraphQL schemas, we'll define our own simple language. We'll use the "GraphQL schema language" - it's similar to the query language, and allows us to talk about GraphQL schemas in a language-agnostic way.

9)

The most basic components of a GraphQL schema are object types, which just represent a kind of object you can fetch from your service, and what fields it has. In the GraphQL schema language, we might represent it like this:

10)

The language is pretty readable, but let's go over it so that we can have a shared vocabulary:

11)

Character is a GraphQL Object Type, meaning it's a type with some fields. Most of the types in your schema will be object types.

12)

name and appearsIn are fields on the Character type. That means that name and appearsIn are the only fields that can appear in any part of a GraphQL query that operates on the Character type.

13)

String is one of the built-in scalar types - these are types that resolve to a single scalar object, and can't have sub-selections in the query. We'll go over scalar types more later.

14)

String! means that the field is non-nullable, meaning that the GraphQL service promises to always give you a value when you query this field. In the type language, we'll represent those with an exclamation mark.

15)

Now you know what a GraphQL object type looks like, and how to read the basics of the GraphQL type language.

16)

Every field on a GraphQL object type can have zero or more arguments, for example the length field

below:

17)

All arguments are named. Unlike languages like JavaScript and Python where functions take a list of ordered arguments, all arguments in GraphQL are passed by name specifically. In this case, the length field has one defined argument, unit.

18)

That means that the GraphQL service needs to have a Query type with hero and droid fields:

19)

Mutations work in a similar way - you define fields on the Mutation type, and those are available as the root mutation fields you can call in your query.

20)

It's important to remember that other than the special status of being the "entry point" into the schema, the Query and Mutation types are the same as any other GraphQL object type, and their fields work exactly the same way.

21)

A GraphQL object type has a name and fields, but at some point those fields have to resolve to some concrete data. That's where the scalar types come in: they represent the leaves of the query.

22)

We know this because those fields don't have any sub-fields - they are the leaves of the query.

23)

GraphQL comes with a set of default scalar types out of the box:

24)

Int: A signed 32-bit integer.

25)

Float: A signed double-precision floating-point value.

26)

String: A UTF-8 character sequence.

27)

Boolean: true or false.

28)

ID: The ID scalar type represents a unique identifier, often used to refetch an object or as the key for a cache. The ID type is serialized in the same way as a String; however, defining it as an ID signifies that it is not intended to be human-readable.

29)

In most GraphQL service implementations, there is also a way to specify custom scalar types. For example, we could define a Date type:

30)

Then it's up to our implementation to define how that type should be serialized, deserialized, and validated. For example, you could specify that the Date type should always be serialized into an integer timestamp, and your client should know to expect that format for any date fields.

31)

Also called Enums, enumeration types are a special kind of scalar that is restricted to a particular set of allowed values. This allows you to:

32)

Validate that any arguments of this type are one of the allowed values

33)

Communicate through the type system that a field will always be one of a finite set of values

34)

Here's what an enum definition might look like in the GraphQL schema language:

35)

This means that wherever we use the type Episode in our schema, we expect it to be exactly one of NEWHOPE, EMPIRE, or JEDI.

36)

Note that GraphQL service implementations in various languages will have their own language-specific way to deal with enums. In languages that support enums as a first-class citizen, the

implementation might take advantage of that; in a language like JavaScript with no enum support, these values might be internally mapped to a set of integers. However, these details don't leak out to the client, which can operate entirely in terms of the string names of the enum values.

37)

Object types, scalars, and enums are the only kinds of types you can define in GraphQL. But when you use the types in other parts of the schema, or in your query variable declarations, you can apply additional type modifiers that affect validation of those values. Let's look at an example:

38)

Here, we're using a String type and marking it as Non-Null by adding an exclamation mark, ! after the type name. This means that our server always expects to return a non-null value for this field, and if it ends up getting a null value that will actually trigger a GraphQL execution error, letting the client know that something has gone wrong.

39)

The Non-Null type modifier can also be used when defining arguments for a field, which will cause the GraphQL server to return a validation error if a null value is passed as that argument, whether in the GraphQL string or in the variables.

40)

Lists work in a similar way: We can use a type modifier to mark a type as a List, which indicates that this field will return an array of that type. In the schema language, this is denoted by wrapping the type in square brackets, [and]. It works the same for arguments, where the validation step will expect an array for that value.

41)

The Non-Null and List modifiers can be combined. For example, you can have a List of Non-Null Strings:

42)

This means that the list itself can be null, but it can't have any null members. For example, in JSON:

43)

Now, let's say we defined a Non-Null List of Strings:

44)

This means that the list itself cannot be null, but it can contain null values:

45)

You can arbitrarily nest any number of Non-Null and List modifiers, according to your needs.

46)

Like many type systems, GraphQL supports interfaces. An Interface is an abstract type that includes a certain set of fields that a type must include to implement the interface.

47)

For example, you could have an interface Character that represents any character in the Star Wars trilogy:

48)

This means that any type that implements Character needs to have these exact fields, with these arguments and return types.

49)

For example, here are some types that might implement Character:

50)

You can see that both of these types have all of the fields from the Character interface, but also bring in extra fields, totalCredits, starships and primaryFunction, that are specific to that particular type of character.

51)

Interfaces are useful when you want to return an object or set of objects, but those might be of several different types.

52)

For example, note that the following query produces an error:

53)

The hero field returns the type Character, which means it might be either a Human or a Droid

depending on the episode argument. In the query above, you can only ask for fields that exist on the Character interface, which doesn't include primaryFunction.

54)

To ask for a field on a specific object type, you need to use an inline fragment:

55)

Learn more about this in the inline fragments section in the query guide.

56)

Union types are very similar to interfaces, but they don't get to specify any common fields between the types.

57)

Wherever we return a SearchResult type in our schema, we might get a Human, a Droid, or a Starship. Note that members of a union type need to be concrete object types; you can't create a union type out of interfaces or other unions.

58)

In this case, if you query a field that returns the SearchResult union type, you need to use an inline fragment to be able to query any fields at all:

59)

The __typename field resolves to a String which lets you differentiate different data types from each other on the client.

60)

Also, in this case, since Human and Droid share a common interface (Character), you can query their common fields in one place rather than having to repeat the same fields across multiple types:

61)

Note that name is still specified on Starship because otherwise it wouldn't show up in the results given that Starship is not a Character!

62)

So far, we've only talked about passing scalar values, like enums or strings, as arguments into a field. But you can also easily pass complex objects. This is particularly valuable in the case of mutations, where you might want to pass in a whole object to be created. In the GraphQL schema language, input types look exactly the same as regular object types, but with the keyword input instead of type:

63)

Here is how you could use the input object type in a mutation:

64)

The fields on an input object type can themselves refer to input object types, but you can't mix input and output types in your schema. Input object types also can't have arguments on their fields.

From:

<http://jace.link/> - **Various Ways**

Permanent link:

<http://jace.link/open/schemas-and-types>

Last update: **2022/09/01 02:41**

