Schemas and Types

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

이 페이지에서는 GraphQL 유형 시스템과 쿼리할 수 있는 데이터를 설명하는 방법에 대해 알아야 할 모든 것을 배울 것입니다. GraphQL은 모든 백엔드 프레임워크 또는 프로그래밍 언어와 함께 사용할 수 있으므로 구현 관련 세부 사항은 피하고 개념에 대해서만 이야기하겠습니다.[1)]

Type System

이전에 GraphQL 쿼리를 본 적이 있다면 GraphQL 쿼리 언어가 기본적으로 개체의 필드를 선택하는 것에 관한 것임을 알고 있습니다. 예를 들어 다음 쿼리에서:[2)]

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

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

1. 특별한 `root` 객체로 시작합니다.[3)]
2. `hero` 필드를 선택합니다.[4)]
3. 반환된 `hero` 객체에서 `name`과 `appearsIn` 필드를 선택합니다.[5)]

GraphQL 쿼리의 모양은 결과와 거의 일치하기 때문에 서버에 대해 많이 알지 않고도 쿼리가 반환할 내용을 예측할 수 있습니다. 그러나 우리가 요청할 수 있는 데이터에 대한 정확한 설명을 갖는 것이 유용합니다. 어떤 필드를 선택할 수 있습니까? 그들은 어떤 종류의 물건을 반환할 수 있습니까? 해당 하위 개체에서 사용할 수 있는 필드는 무엇입니까? 바로 여기에서 스키마가 등장합니다.[6)]

모든 GraphQL 서비스는 해당 서비스에서 쿼리할 수 있는 가능한 데이터 집합을 완전히 설명하는 유형 집합을 정의합니다. 그런 다음 쿼리가 들어오면 해당 스키마에 대해 유효성이 검사되고 실행됩니다.[7)]

Type Language

GraphQL 서비스는 모든 언어로 작성할 수 있습니다. GraphQL 스키마에 대해 이야기하기 위해 JavaScript와 같은 특정 프로그래밍 언어 구문에 의존할 수 없기 때문에 우리만의 간단한 언어를 정의하겠습니다. “GraphQL 스키마 언어 ” 를 사용하겠습니다. 쿼리 언어와 유사하며 언어에 구애받지 않는 방식으로 GraphQL 스키마에 대해 이야기할 수 있습니다.[8)]

Object Types and Fields

GraphQL 스키마의 가장 기본적인 구성 요소는 서비스에서 가져올 수 있는 개체의 종류와 포함하는 필드를 나타내는 개체 유형입니다. GraphQL 스키마 언어에서는 다음과 같이 나타낼 수 있습니다.[9)]

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

언어는 꽤 읽기 쉽지만 공유 어휘를 가질 수 있도록 살펴 보겠습니다.[10)]

- Character는 GraphQL Object Type이며, 이는 일부 필드가 있는 유형임을 의미합니다. 스키마에 있는 대부분의 유형은 객체 유형이 됩니다.[11)]
- name및 appearsIn은 Character타입의 필드 입니다. 즉 name 및 appearsIn은 Character 유형에서 작동하는 GraphQL 쿼리의 모든 부분에 나타날 수 있는 유일한 필드입니다.[12)]
- String은 내장 스칼라 유형 중 하나입니다. 단일 스칼라 객체로 해석되는 유형이며 쿼리에서 하위 선택을 가질 수 없습니다. 스칼라 유형에 대해서는 나중에 더 살펴보겠습니다.[13)]
- String!은 필드가 non-nullable임을 의미합니다. 즉, GraphQL 서비스는 이 필드를 쿼리할 때 항상 값을 제공할 것을 약속합니다. type language에서는 느낌표로 표시합니다.[14)]
- [Episode!]!은 Episode객체의 배열을 나타냅니다. Episode 또한 non-nullable이므로 appearsIn 필드를 쿼리할 때 항상 배열(0개 이상의 항목 포함)을 기대할 수 있습니다. 또한 Episode!은 non-nullable이므로 항상 배열의 모든 항목이 Episode 개체일 것으로 예상할 수 있습니다.

이제 GraphQL 객체 유형이 어떻게 생겼는지, GraphQL 유형 언어의 기본 사항을 읽는 방법을 알게 되었습니다.[15)]

Arguments

GraphQL 객체 유형의 모든 필드는 0개 이상의 인수를 가질 수 있습니다(예, 아래의 length 필드)[16)]

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

모든 인수의 이름이 지정됩니다. 함수가 순서가 지정된 인수 목록을 취하는 JavaScript 및 Python과 같은 언어와 달리 GraphQL의 모든 인수는 구체적으로 이름으로 전달됩니다. 이 경우 length 필드에는 정의된 하나의 unit 인수가 있습니다.[17)]

The Query and Mutation Types

스키마의 대부분의 유형은 일반 객체 유형이지만 스키마 내에는 특별한 두 가지 유형이 있습니다.

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

모든 GraphQL 서비스에는 query유형이 있으며 mutation유형이 있을 수도 있고 없을 수도 있습니다. 이러한 유형은 일반 개체 유형과 동일하지만, 모든 GraphQL 쿼리의 진입점을 정의하기 때문에 특별합니다. 따라서 다음과 같은 쿼리가 표시되면

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

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

즉, GraphQL 서비스에는 hero 및 droid 필드가 있는 Query 유형이 있어야 합니다.[18)]

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

Mutations 은 유사한 방식으로 작동합니다. Mutation유형에 필드를 정의하면 쿼리에서 호출할 수 있는 root mutation 필드로 사용할 수 있습니다.[19)]

스키마에 대한 " 진입점 " 이라는 특수한 상태를 제외하고Query및 Mutation유형은 다른 GraphQL 객체 유형과 동일하며 해당 필드는 정확히 동일한 방식으로 작동한다는 점을 기억하는 것이 중요합니다.[20)]

Scala Types

GraphQL 객체 유형에는 이름과 필드가 있지만 어느 시점에서 이러한 필드는 구체적인 데이터로 해석되어야 합니다. 그것이 스칼라 유형이 들어오는 곳입니다. 그들은 쿼리의 잎을 나타냅니다.[21)]

다음 쿼리에서 `name` 및 `appearsIn` 필드는 스칼라 유형으로 확인됩니다.

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

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

이러한 필드에는 하위 필드가 없기 때문에 이를 알 수 있습니다. 이는 쿼리의 잎사귀이기 때문입니다.[22)]

GraphQL은 기본 스칼라 유형 세트와 함께 제공됩니다.[23)]

- Int: 부호 있는 32비트 정수입니다.[24)]
- Float: 부호 있는 배정밀도 부동 소수점 값입니다.[25)]
- String: UTF-8 문자 시퀀스입니다.[26)]
- Boolean: true 또는 false.[27)]
- ID: ID 스칼라 유형은 개체를 다시 가져오거나 캐시의 키로 자주 사용되는 고유 식별자를 나타냅니다. ID 유형은 문자열과 같은 방식으로 직렬화됩니다. 그러나 ID로 정의하는 것은 사람이 읽을 수 있도록 의도되지 않았음을 의미합니다.[28)]

대부분의 GraphQL 서비스 구현에는 사용자 지정 스칼라 유형을 지정하는 방법도 있습니다. 예를 들어, Date 유형을 정의할 수 있습니다.[29)]

```
scalar Date
```

그런 다음 해당 유형을 직렬화, 역직렬화 및 검증하는 방법을 정의하는 것은 구현에 달려 있습니다. 예를 들어 Date 유형이 항상 정수 타임스탬프로 직렬화되도록 지정할 수 있으며 클라이언트는 모든 날짜 필드에 대해 해당 형식을 예상해야 합니다.[30)]

Enumeration Types

Enums 이라고도 하는 열거형 유형은 특정 허용 값 집합으로 제한되는 특수한 종류의 스칼라입니다. 이를 통해 다음을 수행할 수 있습니다.[31)]

1. 이 유형의 모든 인수가 허용되는 값 중 하나인지 확인한다.[32)]
2. 필드가 항상 유한한 값 집합 중 하나라는 것을 유형 시스템을 통해 전달한다.[33)]

GraphQL 스키마 언어에서 열거형 정의는 다음과 같습니다.[34)]

```
enum Episode {
  NEWHOPE
```

```
  EMPIRE
  JEDI
}
```

이것은 스키마에서 Episode 유형을 사용할 때마다 정확히 NEWHOPE, EMPIRE 또는 JEDI 중 하나일 것으로 예상한다는 것을 의미합니다.[35)]

다양한 언어로 된 GraphQL 서비스 구현에는 열거형을 처리하는 고유한 언어별 방법이 있습니다. 열거형을 일급 시민으로 지원하는 언어에서 구현은 이를 활용할 수 있습니다. 열거형을 지원하지 않는 JavaScript와 같은 언어에서 이러한 값은 내부적으로 정수 집합에 매핑될 수 있습니다. 그러나 이러한 세부 사항은 열거형 값의 문자열 이름 측면에서 완전히 작동할 수 있는 클라이언트로 누출되지 않습니다.[36)]

Lists and Non-Null

개체 유형, 스칼라 및 열거형은 GraphQL에서 정의할 수 있는 유일한 유형의 유형입니다. 그러나 스키마의 다른 부분이나 쿼리 변수 선언에서 형식을 사용 하는 경우 해당 값의 유효성 검사에 영향을 주는 추가 형식 수정자를 적용할 수 있습니다. 예를 살펴보겠습니다.[37)]

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

여기서는 String유형을 사용하고 유형 이름 뒤에 느낌표 !를 추가하여 Non-Null로 표시합니다. 이것은 우리 서버가 항상 이 필드에 대해 null이 아닌 값을 반환할 것으로 예상하고 실제로 GraphQL 실행 오류를 트리거하는 null 값을 얻게 되면 클라이언트에게 문제가 있음을 알릴 것을 기대한다는 것을 의미합니다.[38)]

Null이 아닌 유형 수정자는 필드에 대한 인수를 정의할 때도 사용할 수 있습니다. 그러면 GraphQL 문자열이든 변수든 상관없이 해당 인수로 null 값이 전달되면 GraphQL 서버가 유효성 검사 오류를 반환합니다.[39)]

```
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
        }
      ]
    }
```

```
  ]
}
```

목록은 비슷한 방식으로 작동합니다. 유형 수정자를 사용하여 유형을 목록으로 표시할 수 있습니다. 이는 이 필드가 해당 유형의 배열을 반환함을 나타냅니다. 스키마 언어에서 이는 유형을 대괄호 [및]로 묶어 표시합니다. 유효성 검사 단계에서 해당 값에 대한 배열을 예상하는 인수에 대해 동일하게 작동합니다.[40)]

Non-Null 및 List 수정자를 결합할 수 있습니다. 예를 들어 Null이 아닌 문자열 목록이 있을 수 있습니다.[41)]

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

즉, 목록 자체는 null일 수 있지만 null 멤버는 포함할 수 없습니다. 예를 들어 JSON에서:[42)]

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

이제 null이 아닌 문자열 목록을 정의했다고 가정해 보겠습니다.[43)]

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

이는 목록 자체가 null일 수 없지만 null 값을 포함할 수 있음을 의미합니다.[44)]

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

필요에 따라 Non-Null 및 List 수정자를 임의로 중첩할 수 있습니다.[45)]

Interfaces

많은 유형 시스템과 마찬가지로 GraphQL은 인터페이스를 지원합니다. 인터페이스는 유형이 인터페이스를 구현하기 위해 포함해야 하는 특정 필드 세트를 포함하는 추상 유형입니다.[46)]

예를 들어, Star Wars 3부작의 모든 캐릭터를 나타내는 Character 인터페이스가 있을 수 있습니다.[47)]

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

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

즉, Character를 구현하는 모든 유형에서는 인수 및 반환 유형이 정확한 필드가 있어야 합니다.[48)]

예를 들어 다음은 Character를 구현할 수 있는 몇 가지 유형입니다.[49)]

```
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
}
```

이 두 유형 모두 Character 인터페이스의 모든 필드를 가지고 있지만 특정 유형의 캐릭터와 관련된 추가 필드, `totalCredits`, `starships` 및 `primaryFunction`도 가져오는 것을 볼 수 있습니다.[50)]

인터페이스는 개체 또는 개체 집합을 반환하려는 경우에 유용하지만 여러 유형이 있을 수 있습니다.[51)]

예를 들어 다음 쿼리는 오류를 생성합니다.[52)]

```
query HeroForEpisode($ep: Episode!)
{
  hero(episode: $ep) {
    name
    primaryFunction
  }
}
```

VARIABLES

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

```
{
  "errors": [
    {
      "message": "Cannot query
field \"primaryFunction\" on type
\"Character\". Did you mean to use
an inline fragment on \"Droid\"?",
      "locations": [
        {
          "line": 4,
          "column": 5
        }
      ]
    }
```

```
  ]
}
```

Hero 필드는 Character 유형을 반환합니다. 즉, 에피소드 인수에 따라 Human 또는 Droid가 될 수 있습니다. 위 쿼리에서 PrimaryFunction을 포함하지 않는 Character 인터페이스에 있는 필드만 요청할 수 있습니다.[53)]

특정 객체 유형에 대한 필드를 요청하려면 인라인 조각을 사용해야 합니다.[54)]

```
query HeroForEpisode($ep: Episode!)
{
  hero(episode: $ep) {
    name
    ... on Droid {
      primaryFunction
    }
  }
}
```

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

VARIABLES

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

쿼리 가이드의 인라인 조각 섹션에서 이에 대해 자세히 알아보세요.[55)]

Union Types

Union 유형은 인터페이스와 매우 유사하지만 유형 간에 공통 필드를 지정하지 못합니다.[56)]

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

스키마에서 `SearchResult` 유형을 반환할 때마다 `Human`, `Droid` 또는 `Starship`을 얻을 수 있습니다. 공용체 유형의 구성원은 구체적인 객체 유형이어야 합니다. 인터페이스 또는 다른 공용체에서 공용체 유형을 만들 수 없습니다.[57)]

이 경우 SearchResult 공용체 유형을 반환하는 필드를 쿼리하는 경우 인라인 프래그먼트를 사용해야 모든 필드를 쿼리할 수 있습니다.[58)]

```
{
  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 필드는 클라이언트에서 서로 다른 데이터 유형을 구별할 수 있는 문자열로 확인됩니다.[59)]

또한 이 경우 Human과 Droid는 공통 인터페이스(Character)를 공유하므로 여러 유형에 걸쳐 동일한 필드를 반복할 필요 없이 공통 필드를 한 곳에서 쿼리할 수 있습니다.[60)]

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

이름은 여전히 Starship에 지정되어 있습니다. 그렇지 않으면 Starship이 캐릭터가 아닌 경우 결과에 표시되지 않기 때문입니다![61)]

Input Types

지금까지는 열거형이나 문자열과 같은 스칼라 값을 필드에 인수로 전달하는 것에 대해서만 이야기했습니다. 그러나 복잡한 객체를 쉽게 전달할 수도 있습니다. 이것은 생성할 전체 객체를 전달하려는 mutations의 경우에 특히 유용합니다. GraphQL 스키마 언어에서 입력 유형은 일반 객체 유형과 완전히 동일하지만 유형 대신 `input` 키워드를 사용합니다.[62)]

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

mutation에서 입력 객체 유형을 사용하는 방법은 다음과 같습니다.[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!"
  }
}
```

입력 개체 유형의 필드는 자체적으로 입력 개체 유형을 참조할 수 있지만 스키마에서 입력 및 출력 유형을 혼합할 수는 없습니다. 또한 입력 개체 유형은 해당 필드에 인수를 가질 수 없습니다.[64)]

Continue Reading

- Validation

관련 문서

Plugin Backlinks: 아무 것도 없습니다.

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.