# GraphQL Introspection

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

GraphQL 스키마가 지원하는 쿼리에 대한 정보를 요청하는 것이 종종 유용합니다. GraphQL을 사용하면 내성 시스템을 사용하여 그렇게 할 수 있습니다![1)]

Star Wars 예제의 경우 starWarsIntrospection-test.ts 파일에는 자체 검사 시스템을 보여주는 여러 쿼리가 포함되어 있으며 참조 구현의 자체 검사 시스템을 실행하기 위해 실행할 수 있는 테스트 파일입니다.[2)]

우리는 유형 시스템을 설계했기 때문에 어떤 유형이 사용 가능한지 알 수 있지만, 그렇지 않은 경우 쿼리의 루트 유형에서 항상 사용할 수 있는 __schema 필드를 쿼리하여 GraphQL에 요청할 수 있습니다. 이제 그렇게 하고 어떤 유형을 사용할 수 있는지 물어보겠습니다.[3)]

```
{
  __schema {
    types {
      name
    }
  }
}
```

```
{
  "data": {
    "__schema": {
      "types": [
        {
          "name": "Query"
        },
        {
          "name": "String"
        },
        {
          "name": "ID"
        },
        {
          "name": "Mutation"
        },
        {
          "name": "Episode"
        },
        {
          "name": "Character"
        },
        {
          "name": "Int"
        },
        {
          "name": "LengthUnit"
        },
        {
          "name": "Human"
        },
        {
          "name": "Float"
        },
```

```
        {
          "name": "Droid"
        },
        {
          "name":
"FriendsConnection"
        },
        {
          "name": "FriendsEdge"
        },
        {
          "name": "PageInfo"
        },
        {
          "name": "Boolean"
        },
        {
          "name": "Review"
        },
        {
          "name": "ReviewInput"
        },
        {
          "name": "Starship"
        },
        {
          "name": "SearchResult"
        },
        {
          "name": "__Schema"
        },
        {
          "name": "__Type"
        },
        {
          "name": "__TypeKind"
        },
        {
          "name": "__Field"
        },
        {
          "name": "__InputValue"
        },
        {
          "name": "__EnumValue"
        },
        {
          "name": "__Directive"
        },
        {
          "name":
```

```
"__DirectiveLocation"
        }
      ]
    }
  }
}
```

## Group Types

우와 종류가 많네요! 그들은 무엇인가? 그룹화해 보겠습니다.[4)]

- Query, Character, Human, Episode, Droid : 이것들은 우리의 유형 시스템에서 정의한 것들입니다.[5)]
- String, Boolean : 타입 시스템이 제공한 내장 스칼라입니다.[6)]
- `__Schema, __Type, __TypeKind, __Field, __InputValue, __EnumValue, __Directive` : 이 모든 항목 앞에는 이중 밑줄이 표시되어 내부 검사 시스템의 일부임을 나타냅니다.[7)]

이제 어떤 쿼리를 사용할 수 있는지 살펴보기에 좋은 위치를 찾아 보겠습니다. 유형 시스템을 설계할 때 모든 쿼리가 시작되는 유형을 지정했습니다. 그것에 대해 내성 시스템에 물어보자![8)]

```
{
  __schema {
    queryType {
      name
    }
  }
}
```

```
{
  "data": {
    "__schema": {
      "queryType": {
        "name": "Query"
      }
    }
  }
}
```

그리고 그것은 우리가 유형 시스템 섹션에서 말한 것과 일치합니다. 쿼리 유형은 우리가 시작할 곳입니다! 여기에서 명명한 것은 관례에 따른 것입니다. 쿼리 유형의 이름을 다른 이름으로 지정할 수 있으며 쿼리의 시작 유형으로 지정했다면 여전히 여기에 반환되었을 것입니다. 하지만 이름을 Query로 지정하는 것은 유용한 규칙입니다.[9)]

하나의 특정 유형을 조사하는 것이 종종 유용합니다. Droid 유형을 살펴보겠습니다.[10)]

```
{
  __type(name: "Droid") {
    name
  }
}
```

```
{
  "data": {
    "__type": {
      "name": "Droid"
    }
  }
}
```

하지만 Droid에 대해 더 알고 싶다면 어떻게 해야 할까요? 예를 들어 인터페이스인가 객체인가?[11)]

```
{
  __type(name: "Droid") {
    name
    kind
  }
}
```

```
{
  "data": {
    "__type": {
      "name": "Droid",
      "kind": "OBJECT"
    }
  }
}
```

kind는 값 중 하나가 OBJECT인 __TypeKind 열거형을 반환합니다. 대신 Character에 대해 물으면 인터페이스라는 것을 알 수 있습니다.[12)]

```
{
  __type(name: "Character") {
    name
    kind
  }
}
```

```
{
  "data": {
    "__type": {
      "name": "Character",
      "kind": "INTERFACE"
    }
  }
}
```

객체가 어떤 필드를 사용할 수 있는지 아는 것이 유용하므로 Droid에 대한 내성 시스템에 물어보겠습니다.[13)]

```
{
  __type(name: "Droid") {
    name
    fields {
      name
      type {
        name
        kind
      }
    }
  }
}
```

```
{
  "data": {
    "__type": {
      "name": "Droid",
      "fields": [
        {
          "name": "id",
          "type": {
            "name": null,
            "kind": "NON_NULL"
          }
        },
        {
          "name": "name",
          "type": {
            "name": null,
            "kind": "NON_NULL"
          }
```

```
        },
        {
          "name": "friends",
          "type": {
            "name": null,
            "kind": "LIST"
          }
        },
        {
          "name":
"friendsConnection",
          "type": {
            "name": null,
            "kind": "NON_NULL"
          }
        },
        {
          "name": "appearsIn",
          "type": {
            "name": null,
            "kind": "NON_NULL"
          }
        },
        {
          "name":
"primaryFunction",
          "type": {
            "name": "String",
            "kind": "SCALAR"
          }
        }
      ]
    }
  }
}
```

그것들은 우리가 Droid에서 정의한 우리의 필드입니다![14)]

id는 거기에서 약간 이상해 보입니다. 유형에 대한 이름이 없습니다. NON_NULL 종류의 “ 래퍼 ” 유형이기 때문입니다. 해당 필드의 유형에 대해 ofType을 쿼리하면 ID 유형이 발견되어 이것이 null이 아닌 ID임을 알려줍니다.[15)]

유사하게, friend와 appearsIn은 모두 LIST 래퍼 유형이기 때문에 이름이 없습니다. 우리는 이러한 유형에 대해 ofType을 쿼리할 수 있으며 이는 목록이 무엇인지 알려줍니다.[16)]

```
{
  __type(name: "Droid") {
    name
    fields {
```

```
{
  __type(name: "Droid") {
    name
    fields {
```

```
      name
      type {
        name
        kind
        ofType {
          name
          kind
        }
      }
    }
  }
}
```

```
      name
      type {
        name
        kind
        ofType {
          name
          kind
        }
      }
    }
  }
}
```

툴링에 특히 유용한 인트로스펙션 시스템의 기능으로 끝내자. 시스템에 문서를 요청합시다![17)]

```
{
  __type(name: "Droid") {
    name
    description
  }
}
```

```
{
  "data": {
    "__type": {
      "name": "Droid",
      "description": null
    }
  }
}
```

따라서 우리는 내성을 사용하여 유형 시스템에 대한 문서에 액세스하고 문서 브라우저 또는 풍부한 IDE 환경을 만들 수 있습니다.[18)]

이것은 자기 성찰 시스템의 표면을 긁은 것뿐입니다. 열거형 값, 유형이 구현하는 인터페이스 등을 쿼리할 수 있습니다. 우리는 자기 성찰 시스템 자체를 성찰할 수도 있습니다. 사양은 “ 인트로스펙션 ” 섹션에서 이 주제에 대해 더 자세히 설명하고, GraphQL.js의 내성 파일에는 사양을 준수하는 GraphQL 쿼리 내성 시스템을 구현하는 코드가 포함되어 있습니다.[19)]

## Continue Reading

- GraphQL Best Practices

---

## 관련 문서

**Plugin Backlinks: 아무 것도 없습니다.**

1)

It's often useful to ask a GraphQL schema for information about what queries it supports. GraphQL allows us to do so using the introspection system!

2)

For our Star Wars example, the file starWarsIntrospection-test.ts contains a number of queries demonstrating the introspection system, and is a test file that can be run to exercise the reference

implementation's introspection system.
3)

We designed the type system, so we know what types are available, but if we didn't, we can ask GraphQL, by querying the `__schema` field, always available on the root type of a Query. Let's do so now, and ask what types are available.
4)

Wow, that's a lot of types! What are they? Let's group them:
5)

These are the ones that we defined in our type system.
6)

These are built-in scalars that the type system provided.
7)

These all are preceded with a double underscore, indicating that they are part of the introspection system.
8)

Now, let's try and figure out a good place to start exploring what queries are available. When we designed our type system, we specified what type all queries would start at; let's ask the introspection system about that!
9)

And that matches what we said in the type system section, that the Query type is where we will start! Note that the naming here was just by convention; we could have named our Query type anything else, and it still would have been returned here had we specified it was the starting type for queries. Naming it Query, though, is a useful convention.
10)

It is often useful to examine one specific type. Let's take a look at the Droid type:
11)

What if we want to know more about Droid, though? For example, is it an interface or an object?
12)

kind returns a `__TypeKind` enum, one of whose values is OBJECT. If we asked about Character instead we'd find that it is an interface:
13)

It's useful for an object to know what fields are available, so let's ask the introspection system about Droid:
14)

Those are our fields that we defined on Droid!
15)

id looks a bit weird there, it has no name for the type. That's because it's a “wrapper” type of kind `NON_NULL`. If we queried for ofType on that field's type, we would find the ID type there, telling us that this is a non-null ID.
16)

Similarly, both friends and appearsIn have no name, since they are the LIST wrapper type. We can query for ofType on those types, which will tell us what these are lists of.
17)

Let's end with a feature of the introspection system particularly useful for tooling; let's ask the system for documentation!
18)

So we can access the documentation about the type system using introspection, and create documentation browsers, or rich IDE experiences.
19)

This has just scratched the surface of the introspection system; we can query for enum values, what interfaces a type implements, and more. We can even introspect on the introspection system itself. The specification goes into more detail about this topic in the “Introspection” section, and the introspection file in GraphQL.js contains code implementing a specification-compliant GraphQL query introspection system.