GraphQL Execution

검증된 후 GraphQL 쿼리는 일반적으로 JSON으로 요청된 쿼리의 모양을 미러링하는 결과를 반환하는 GraphQL 서버에 의해 실행됩니다.[1)]

GraphQL은 타입 시스템 없이 쿼리를 실행할 수 없습니다. 쿼리 실행을 설명하기 위해 예제 타입 시스템을 사용하겠습니다. 이것은 다음 기사의 예제 전체에서 사용된 동일한 타입 시스템의 일부입니다.[2)]

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

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

enum Episode {
  NEWHOPE
  EMPIRE
  JEDI
}

type Starship {
  name: String
}
```

쿼리가 실행될 때 어떤 일이 발생하는지 설명하기 위해 예제를 통해 살펴보겠습니다.[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 쿼리의 각 필드는 다음 유형을 반환하는 이전 유형의 함수 또는 메서드로 생각할 수 있습니다. 사실 이것이 바로 GraphQL이 작동하는 방식입니다. 각 유형의 각 필드는 GraphQL 서버 개발자가 제공하는 리졸버라는 함수에 의해 지원됩니다. 필드가 실행되면 해당 리졸버가 호출되어 다음 값을 생성합니다.[4)]

필드가 문자열이나 숫자와 같은 스칼라 값을 생성하면 실행이 완료됩니다. 그러나 필드가 개체 값을 생성하는 경우 쿼리에는 해당 개체에 적용되는 다른 필드 선택이 포함됩니다. 이것은 스칼라 값에 도달할 때까지 계속됩니다. GraphQL 쿼리는 항상 스칼라 값에서 끝납니다.[5)]

Root fields & resolvers

모든 GraphQL 서버의 최상위 수준에는 GraphQL API에 대한 모든 가능한 진입점을 나타내는 유형이 있으며, 이를 종종 루트 타입 또는 쿼리 타입이라고 합니다.[6)]

이 예에서 쿼리 유형은 id 인수를 허용하는 human 이라는 필드를 제공합니다. 이 필드에 대한 해석기 함수는 데이터베이스에 액세스한 다음 Human 개체를 구성하고 반환합니다.[7)]

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

이 예제는 JavaScript로 작성되었지만 GraphQL 서버는 다양한 언어로 구축할 수 있습니다. 리졸버 함수는 4개의 인수를 받습니다.[8)]

- `obj`: 루트 쿼리 유형의 필드에 대해 자주 사용되지 않는 이전 개체입니다.[9)]
- `args`: GraphQL 쿼리의 필드에 제공된 인수입니다.[10)]
- `context`: 모든 리졸버에 제공되며 현재 로그인한 사용자 또는 데이터베이스에 대한 액세스와 같은 중요한 컨텍스트 정보를 보유하는 값입니다.[11)]
- `info`: 현재 쿼리와 관련된 필드별 정보 및 스키마 세부 정보를 담고 있는 값이며 자세한 내용은 GraphQLResolveInfo 유형을 참조하십시오.[12)]

Asynchronous resolvers

이 리졸버 기능에서 무슨 일이 일어나고 있는지 자세히 살펴보겠습니다.[13)]

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

컨텍스트는 GraphQL 쿼리에서 인수로 제공된 id로 사용자의 데이터를 로드하는 데 사용되는 데이터베이스에 대한 액세스를 제공하는 데 사용됩니다. 데이터베이스에서 로드하는 것은 비동기 작업이므로 Promise를 반환합니다. JavaScript에서 Promise는 비동기식 값으로 작업하는 데 사용되지만 Future, Tasks 또는 Deferred라고 하는 여러 언어에 동일한 개념이 존재합니다. 데이터베이스가 반환되면 새로운 Human 객체를 생성하고 반환할 수 있습니다.[14)]

리졸버 함수는 Promises을 알아야 하지만 GraphQL 쿼리는 그렇지 않습니다. 그것은 단순히 human 필드가 이름을 물어볼 수 있는 무언가를 반환하기를 기대합니다. 실행하는 동안 GraphQL은 계속하기 전에 Promises, Futures 및 Tasks가 완료될 때까지 기다리며 최적의 동시성을 사용합니다.[15)]

Trivial resolvers

이제 Human 개체를 사용할 수 있으므로 GraphQL 실행은 요청된 필드로 계속할 수 있습니다.[16)]

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

GraphQL 서버는 다음에 수행할 작업을 결정하는 데 사용되는 타입 시스템에 의해 구동됩니다. human 필드가 아무것이나 반환하기 전에 GraphQL은 휴먼 필드가 휴먼을 리턴한다고 타입 시스템이 알려 주기 때문에 다음 단계는 Human 타입의 필드를 해결하는 것임을 알고 있습니다.[17)]

이 경우 이름을 확인하는 것은 매우 간단합니다. 이름 확인자 함수가 호출되고 obj 인수는 이전 필드에서 반환된 새 Human 개체입니다. 이 경우 Human 객체에는 직접 읽고 반환할 수 있는 name 속성이 있어야 합니다.[18)]

사실, 많은 GraphQL 라이브러리에서 이렇게 간단한 리졸버를 생략할 수 있으며 필드에 리졸버가 제공되지 않으면 동일한 이름의 속성을 읽고 반환해야 한다고 가정합니다.[19)]

Scalar coercion

name 필드가 해결되는 동안 appearsIn 및 starships 필드는 동시에 해결될 수 있습니다. appearsIn 필드에는 사소한 해석기가 있을 수도 있지만 자세히 살펴보겠습니다.[20)]

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

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

유형 시스템 클레임이 나타납니다. 값이 알려진 열거형 값을 반환하지만 이 함수는 숫자를 반환합니다! 실제로 결과를 보면 적절한 Enum 값이 반환되고 있음을 알 수 있습니다. 무슨 일이야?[21)]

이것은 스칼라 강제 변환의 예입니다. 유형 시스템은 무엇을 예상해야 하는지 알고 있으며 리졸버 함수에서 반환된 값을 API 계약을 유지하는 것으로 변환합니다. 이 경우 내부적으로 4, 5, 6과 같은 숫자를 사용하지만 GraphQL 유형 시스템에서 Enum 값으로 나타내는 Enum이 서버에 정의되어 있을 수 있습니다.[22)]

List resolvers

필드가 위의 appearsIn 필드가 있는 항목 목록을 반환할 때 어떤 일이 발생하는지 이미 살펴보았습니다. 열거형 값 목록을 반환했으며 이것이 유형 시스템이 예상한 것이기 때문에 목록의 각 항목은 적절한 열거형 값으로 강제 변환되었습니다. starships 필드가 해결되면 어떻게 됩니까?[23)]

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

이 필드의 해석기는 단지 약속을 반환하는 것이 아니라 약속의 목록을 반환합니다. Human 개체에는 그들이 조종한 Starships의 ID 목록이 있었지만 실제 Starship 개체를 얻으려면 해당 ID를 모두 로드해야 합니다.[24)]

GraphQL은 계속하기 전에 이러한 모든 약속을 동시에 기다리며, 객체 목록이 남으면 이러한 각 항목에 name 필드를 로드하기 위해 동시에 다시 계속됩니다.[25)]

Producing the result

각 필드가 확인되면 결과 값은 필드 이름(또는 별칭)을 키로, 확인된 값을 값으로 사용하여 키-값 맵에 배치됩니다. 이것은 쿼리의 맨 아래 리프 필드에서 루트 쿼리 유형의 원래 필드까지 계속됩니다. 집합적으로 이것들은 원래 쿼리를 미러링하는 구조를 생성한 다음 요청한 클라이언트에 보낼 수 있습니다(일반적으로 JSON으로).[26)]

이 모든 해결 함수가 결과를 생성하는 방법을 확인하기 위해 원래 쿼리를 마지막으로 살펴보겠습니다.[27)]

```
{
  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

관련 문서

Plugin Backlinks: 아무 것도 없습니다.

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)

모든 리졸버에 제공되며 현재 로그인한 사용자 또는 데이터베이스에 대한 액세스와 같은 중요한 컨텍스트 정보를 보유하는 값입니다.
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: