

# GraphQL

GraphQL로 영화 API 만들기

이시영

# Recap

**Why ,Types, Nullable-Field**

# Why OverFetching

- 표현(사용)할 정보 이상으로 서버로부터 데이터를 가져온다

```
// over-fetching의 예제
{
  mypage :{
    Name: Baek
    Birthday : Oct.08
    Hobby : Jogging
    ....
  },
  LikePost:{
    Post_uid: 1
    Post_uid: 3
    ...
  },
  JoinClub:{
    ClubName: "Like Run",
    ClubName: "Make Cook"
    ...
  }
}
```

# Why OverFetching

- 표현(사용)할 정보 이상으로  
서버로부터 데이터를 가져온다

이는 곧 통신을 무겁게 만들며  
프론트 개발자에게 혼란을 유발

```
// over-fetching의 예제
{
  mypage :{
    Name: Baek
    Birthday : Oct.08
    Hobby : Jogging
    ....
  },
  LikePost:{
    Post_uid: 1
    Post_uid: 3
    ...
  },
  JoinClub:{
    ClubName: "Like Run",
    ClubName: "Make Cook"
    ...
  }
}
```

# Why

## UnderFetching

- 표현(사용)할 정보 이하로 서버로부터 데이터를 가져온다
- 하나의 EndPoint로는 데이터가 충족되지 않는 경우

```
// under-fetching의 예제
{
    mypage :{
        Name: Baek
        Birthday : Oct.08
        ....
    }
}
```

# Why

## UnderFetching

- 표현(사용)할 정보 이하로 서버로부터 데이터를 가져온다
  - 하나의 EndPoint로는 데이터가 충족되지 않는 경우

이는 곧 N번의 호출을 발생시켜 UX에 불편함을 가져다준다

```
// under-fetching의 예제
{
    mypage :{
        Name: Baek
        Birthday : Oct.08
        ....
    }
}
```

# Types

## Scaler - 기본(원시) 타입

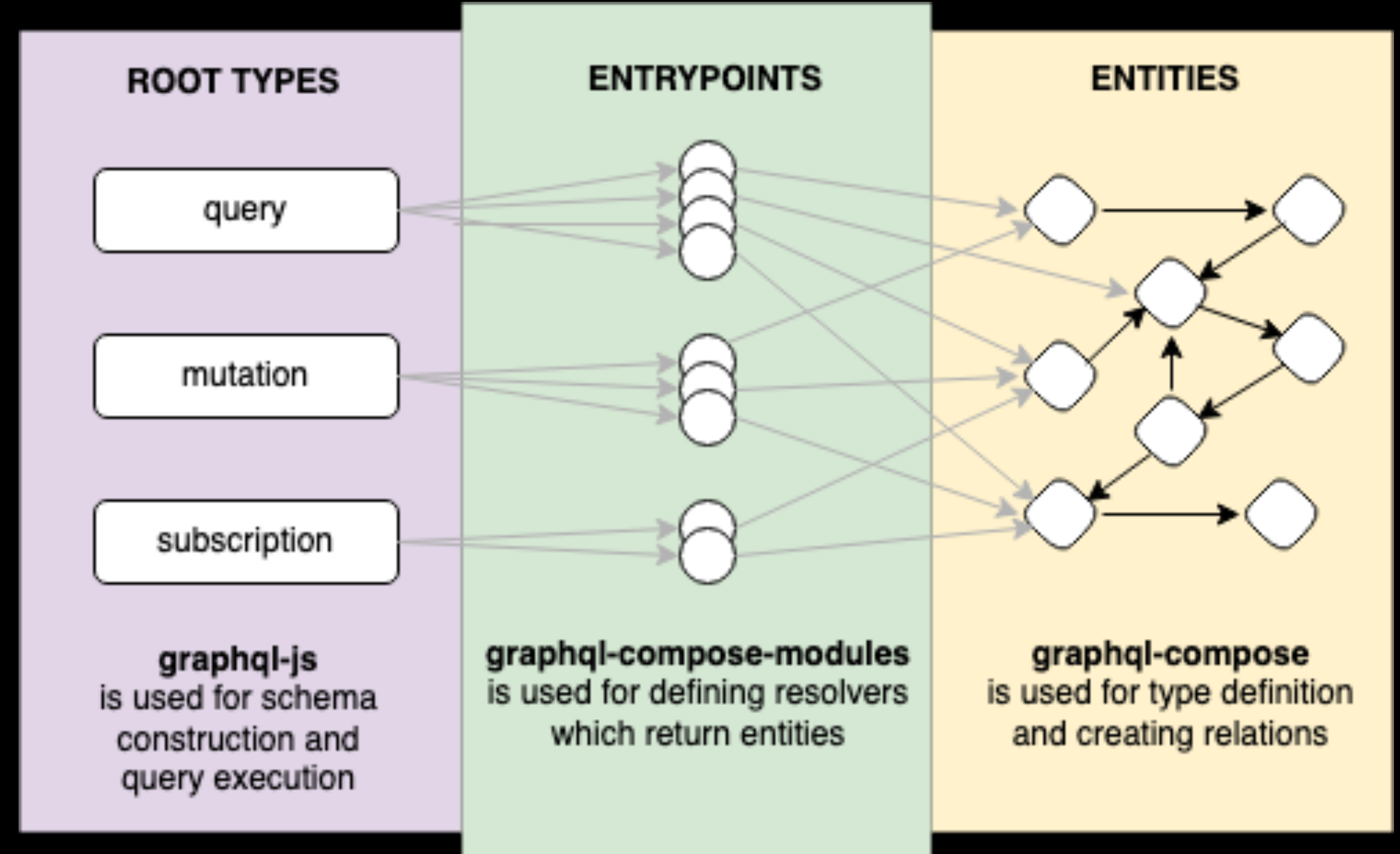
- ID
- Int
- Float
- String
- Boolean

```
type Person {  
    studentID :ID  
    name: String  
    age: Int  
    isStudent: Boolean  
    weight: Float  
}
```

# Types

## Root - 쿼리 시작 지점

- **Query**
  - GET Method
  - 서버에서 데이터 가져오기
- **Mutation**
  - POST, PUT, DELETE Method
  - 서버에 데이터 업데이트하기





# Types

## Mutation Type( in RootType)

- 클라이언트가 데이터를 서버에 새로 올리거나, 데이터를 삭제하거나, 업데이트 하고 싶은 경우

```
type Mutation {  
  postFeed(text: String, userId: ID!): Feed  
  deleteFeed(id: ID!): Boolean  
}
```

# Types

## Mutation Type( in RootType)

- 클라이언트가 데이터를 서버에 새로 올리거나, 데이터를 삭제하거나, 업데이트 하고 싶은 경우

### Operation

```
1  mutation{
2    | postFeed(text: "Hello, first feed", userId: "Joah") {
3    |   | text
4    |   }
5  }
6
```

# Non-Nullable Field

- 절대 Null값을 주고 싶지 않은 경우
  - 인자
  - 반환값
- ! 사용

```
{
  "errors": [
    {
      "message": "Cannot return null for non-nullable
field Query.allFeeds.",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "allFeeds"
      ],
      "extensions": {
        "code": "INTERNAL_SERVER_ERROR",
        "exception": {
          "stacktrace": [
            "Error: Cannot return null for
non-nullable field Query.allFeeds.",
            "    at completeValue (/Users/
eunbyeongkim/Desktop/GraphQL/tweetql/node_modules/
graphql/execution/execute.js:594:13)",
            "    at executeField (/Users/eunbyeongkim/
Desktop/GraphQL/tweetql/node_modules/graphql/
execution/execute.js:489:19)",
            "    at executeFields (/Users/
eunbyeongkim/Desktop/GraphQL/tweetql/node_modules/
graphql/execution/execute.js:413:20)",
            "    at executeOperation (/Users/
```

# Resolvers

Query, Mutation, Type

# GraphQL

## Resolver란

- GraphQL - **Server**에서 **미리 작성해둔 스키마 형식에 맞춰 DB의 데이터를 가공하여 값을 반환해주는 로직 담당**

GraphQL 스키마를 정의했고, 클라이언트에서 온 요청을 서버가 처리하기 위해 필요한 기술을 알아보았습니다. 이제 클라이언트에서 온 요청에 따라 적절한 데이터를 반환하는 과정이 남았습니다. 이는 리졸버(resolver)라는 것이 담당합니다. Java쪽에서는 데이터 페처(data fetcher)라고도 부릅니다. 리졸버를 완전히 이해하면 GraphQL을 전부를 알았다고 할만큼 GraphQL의 핵심이라고 볼 수 있습니다.

# GraphQL

## Resolver 구현 방법

**Type Field** 와 **Resolver**의 함수이름을 반드시 일치시켜주어야한다

```
const resolvers = {
  Query: {
    getPosts: () => posts,
    getUser: (_source: any, args: any) => authors[args.id],
  },
  Post: {
    author: (source: any) => authors[source.author_id],
  },
};
```

```
const type_defs = `
type User {
  id: ID!
  name: String!
  mobile_tel: String!
}

type Post {
  id: ID!
  title: String!
  author_id: ID!
  author: User!
}

type Query {
  getPosts: [Post!]!
  getUser(id: ID!): User!
}`;
```

# Resolver 핵심

- Schema의 이름과 Resolver의 이름이 일치해야한다
- Schema의 타입을 토대로 데이터를 가공하여 반환해야한다
- 인자를 두 개 제공
  - Root
    - 이전 단계의 객체 - 중첩된 객체에서 주로 사용
  - Arguments
    - GraphQL 쿼리에서 전달된 인수

```
const typeDefs = gql`
  type User {
    id: ID!
    username: String!
  }
  type Tweet {
    id: ID!
    text: String!
    author: User!
  }
  type Query {
    allTweets: [Tweet!]!
    tweet(id: ID!): Tweet
  }
  type Mutation {
    postTweet(text: String!, userId: ID!): Tweet!
    deleteTweet(id: ID!): Boolean!
  }
`;

const resolvers = {
  Query: {
    allTweets() {
      return tweets;
    }
    // resolvers function은 아폴로 서버가 해당 함수를 호출할때, 실은 어떤 arguments를 줍니다.
    tweet(root, args) {
      console.log(args);
      return null;
    }
  }
}
```



# Root Type만 가능?

## Type Resolver!!!

- 일반적인 타입 정의도 Resolver를 사용할 수 있습니다
  - 중첩된 객체를 살펴봅시다
    - Query Type의 Tweet 호출
    - Resolver의 Tweet field에서 id, text, author 반환
    - Author는 User type
      - userId를 내려줌으로써(Root 인자) users데이터들에서 해당 유저를 찾아서 반환

```
type User {  
  id: ID!  
  firstName: String!  
  lastName: String!  
  .....  
  Is the sum of firstName + lastName as a string  
  .....  
  fullName: String!  
}  
.....  
Tweet object represents a resource for a Tweet  
.....  
type Tweet {  
  id: ID!  
  text: String!  
  author: User  
}
```

```
User: {  
  fullName({ firstName, lastName }) {  
    return `${firstName} ${lastName}`;  
  },  
},  
Tweet: {  
  author({ userId }) {  
    return users.find((user) => user.id === userId);  
  },  
},
```



# Root Type만 가능?

## Type Resolver!!!

Operation

1

2

3

4

5

6

7

8

```
query Tweet($tweetId: ID!) {  
  tweet(id: $tweetId) {  
    author {  
      id  
      fullName  
    }  
  }  
}
```

...

STATUS 200

```
{  
  "data": {  
    "tweet": {  
      "author": {  
        "id": "1",  
        "fullName": "nico las"  
      }  
    }  
  }  
}
```

Variables

Headers

Pre-Operation Script

Post-Operation Script

1

2

3


```
{  
  "tweetId": "2"  
}
```

JSON

# GraphQL

## Documentation

- ApolloClient나 AltairClient에서 해당 내용이 어떤 데이터를 인자로 줘야하고, 어떤 데이터 유형이 반환되는지 설명을 적는 것

FIELDS	DETAILS	ACTIONS
deleteTweet : <b>Boolean!</b>	Deletes a Tweet if found, else returns false  id <b>ID!</b>	

# 끝

GraphQL 개념을 다 살펴봤습니다