GraphQL Besu 가이드

GraphQL 개요

GraphQL이란

GraphQL은 데이터 질의 및 조작 언어로, API를 통해 데이터를 읽고 쓰는 방법을 제공한다.

GraphQL의 특징으로는 다음과 같이 있다.

- overfetching 해결
  - 클라이언트는 데이터의 정확히 필요한 부분만을 가져올 수 있다.
  - REST API에서 필요 없는 필드들까지 제공받아야 하는 문제를 해결한다.
- underfetching 방지
  - 단일 요청으로 여러 종류의 데이터를 가져올 수 있다.
  - REST API에서 여러 url에 걸친 데이터를 가져오기 위해 여러 요청을 날려야 하는 문제를 해결한다.
- schema-first approach
  - 서버가 반환하는 데이터의 구조를 지정하여, 어떤 요청이 가능하고 요청 시 어떠한 데이터가 반환되는지 명확히 명시한다.
  - schema는 서버와 클라이언트 간의 계약과 같이 동작한다.

GraphQL을 사용하기 위해서는 server 쪽에서 등록한 schema를 확인하여, schema에 맞는 형식으로 요청을 보내야 한다. Besu의 schema에 맞게 요청을 보내기 위해, 사용할 주요 GraphQL 요소들에 대해서 살펴보고자 한다.

[주요 GraphQL Query 요소]

Besu Server에 요청을 보낼 때 사용할 GraphQL Query의 요소들은 다음과 같다.

1. Operation

- 개요
  - Operation은 GraphQL 서버가 노출한 데이터와 상호 작용할 수 있는 방식으로, GraphQL Query의 진입점 역할을 한다.
  - Operation은 Type과 Name을 가진다.
- Operation Type
  - Client가 실행하고자 하는 Operation 종류를 명시한다.
  - Operation Type으로는 query, mutation, subscription이 존재한다.
    - query : 서버로부터 데이터를 읽어오는 요청으로, REST API에서의 GET과 유사하다.

        - mutation : 서버의 데이터를 생성/갱신/삭제하는 요청으로, REST API에서의 POST, PUT, DELETE와 유사하다.
        - subscription : 특정 이벤트 발생 시, 서버에서 Client로 데이터를 보내게 하는 요청으로, 서버와의 실시간 연결을 유지하게 한다.
- Operation Name
    - operation에 의미있고 명시적인 이름을 부여한다.
    - 여러 개의 Operation을 사용할 때만 필수적이지만, 디버깅과 로깅 시 활용을 위해 사용이 권장된다.
- 활용 예시
    - Query 예시

```
{
    "query": "query GetUser{ user { name } }"
}
```

    - Mutation 예시

```
{
    "query": "mutation CreateUser{createUser(name: "james"){name}}"
}
```

2. Fields

- 개요
    - field는 Object가 지니고 있는 값으로, field로 Object type과 scalar type이 올 수 있다.
        - Object type : 하위에 fields를 가지고 있는, server에서 정의한 Type
        - Scalar type : GraphQL 원시 타입으로, `Int`, `Float`, `String`, `Boolean`, `ID`가 있음 (필요에 따라 custom scalar 정의 가능)
    - graphql server에서 데이터를 얻기 위해서는 구체적인 Object의 field들을 명시해야 한다.
        - 서버는 Client가 정확히 어떤 field들을 요청하는지 알아야 한다.
        - GraphQL Query는 얻고자하는 결과와 완벽하게 동일한 형태로 작성되어야 한다.
    - field 지정은 scalr type이 나올 때까지 계속되어야 한다.
        - field가 Object 타입일 경우, 해당 Object의 field 지정이 필요하다
        - scalar type이 GraphQL Query의 leaves, 즉 end-point가 되어야 한다.
- 예시

```
{
    "query" : "{
                hero {
                    name
                    friends {
                        name
                    }
                    }
```

```
                }"
}
------------------------------ 결과
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friends": [
        {
          "name": "Luke Skywalker"
        },
        {
          "name": "Han Solo"
        }
      ]
    }
  }
}
```

3. Arguments

- 개요
  - 상위 Object와 field 모두 인자들을 받을 수 있다.
  - GraphQL 서버에서는 주어진 인자들에 따라 결과를 반환한다.
  - Argument의 타입으로는 Scalar, Object, Enum type이 올 수 있다.
- 예시

```
{
  human(id: "1000") {
    name
    height(unit: FOOT)
  }
}
------------------------
{
  "data": {
    "human": {
      "name": "Luke Skywalker",
      "height": 5.6430448
    }
  }
}
```

4. Aiases

- 개요
  - 요청 시, 결과값으로 받을 field명을 별칭으로 사전에 설정할 수 있다. (optional)
  - 다른 argument를 가진 동일한 field들을 한번에 조회하기 위해서는 Alias를 설정해야 한다. (mandatory)

        - 결과값에는 field의 이름만 들어가고 argument는 포함되지 않기 때문에, 동일한 field를 직접적으로 여러 개 사용할 수 없다
- 예시

```
{
  empireHero: hero(episode: EMPIRE) {
    name
  }
  jediHero: hero(episode: JEDI) {
    name
  }
}
----------------------------------------
{
  "data": {
    "empireHero": {
      "name": "Luke Skywalker"
    },
    "jediHero": {
      "name": "R2-D2"
    }
  }
}
```

5. Variables

- 개요
    - GraphQL은 쿼리에서의 동적인 값을 "variables"라는 별도의 영역에서 정의할 수 있다.
        - 주로, arguments 값들은 동적으로 주어질 것이다.
    - Client는 요청에서 argument 값만 달라질 때, 전체 쿼리를 수정하는 것이 아닌 variables에만 다른 값을 넣으면 된다.
- 사용법
    1. argument로 사용되는 하드코딩된 값을 `$variableName` 형식으로 대체한다.
    2. `$variableName`를 쿼리로 들어오는 하나의 변수로 선언한다.
    3. "variables" 영역에 `variableName: value` 형식으로 들어갈 값을 지정한다.
- 예시

```
{
    "query" : "query ($episode: Episode) {
                    hero(episode: $episode) {
                        name,
                        friends {
                            name
                        }
                    }
                }",
    "variables" : {
            "episode": "JEDI"
            }
}
```

[주요 GraphQL Schema type]

Besu의 GraphQL schema를 이해하기 위해 필요한 GraphQL type들은 다음과 같다.

1. Scalar type

- 개요
  - 쿼리의 끝 부분(leaf)으로, object는 중첩되는 object 구조를 가질 수 있지만, 결국 특정 지점에서는 scalar type으로 귀결되어야 한다.
- 종류
  - `Int`: 부호있는 32-bit 크기의 정수
  - `Float`: 부호있는 실수
  - `String`: UTF-8
  - `Boolean`: `true` or `false`.
  - `ID`: Object의 유일한 구분자로 활용되며, 어떠한 `String` or `Int` 값 사용 가능

2. Object type

- 개요
  - GraphQL 스키마의 가장 기본적인 요소로, 서버에서 제공할 객체와 객체가 가진 field들에 대해서 정의한다.
- 예시

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

  - `Character` : 스키마에서 정의한 Object로, 몇 개의 field들을 가진다.

- `name`, `appearsIn` : `Character` type이 가지는 field이름으로, `:` 이후의 부분은 해당 filed가 반환하는 type이다.
  - `String!`
    - `name` 필드를 지정하면, String 타입의 결과가 반환된다.
    - `!`은 *non-nullable*로, name의 return 값으로는 무조건 String 반환된다.
  - `[Episode!]!`
    - `appearsIn` 필드를 지정하면, `Episode`라는 objects type을 가지는 list가 반환된다.

💡 `[String!]` vs `[String]!`

- `[String!]` : `[]` 내부에 `!`가 있을 때에는 리스트 내부에 `null`이 포함될 수 없음을 의미한다.
- `[String]!` : `[]` 외부에 `!`가 있을 때에는 null 자체로 return 될 수 없음을 의미한다.

	`[String!]`	`[String]!`
null	valid	error
[]	valid	valid
["a", "b"]	valid	valid
["a", null, "b"]	error	valid

3. Input type

- 개요
  - Argument로 사용되는 Object Type
    - Client는 Input Type을 "variables" 영역에서 정의하여 사용가능하며, query로 조회할 수 없다.
  - Object Type과 동일하게 생겼지만, 내부 field들이 agrument를 가질 수 없다
- 예시

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

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

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

4.Enumeration type

- 개요
  - 허용된 값들로 제한된 특별한 종류의 scalar type이다.
  - 다음 2가지 기능을 제공한다.
    1. 해당 Type의 argument가 허용된 값인지 검증한다.
    2. 해당 type의 field가 반환하는 값이 유한한 값들 중 하나임을 보장한다.
- 예시

```
enum Episode {
  NEWHOPE
  EMPIRE
  JEDI
}
```

Besu schema.graphql

오케스트레이터에서 사용할 기능으로는 1.특정 블록 번호에서의 스마트 컨트랙트 데이터 다중 조회 2.다중 서명 트랜잭션 전송 이 있다. 이에, Besu의 schema.graphql도 해당 기능들에 필요한 영역에 대해서만 살펴보고자 한다. 전체 스키마는 아래 링크를 통해 확인 가능하다.

🔗 Besu GraphQL 전체 스키마

schema.graphql 주요 부분

1. Type Query

```
type Query {

  block(number: Long, hash: Bytes32): Block

  blocks(from: Long, to: Long): [Block!]!

  pending: Pending!
```

```
  transaction(hash: Bytes32!): Transaction

  logs(filter: FilterCriteria!): [Log!]!

  gasPrice: BigInt!

  maxPriorityFeePerGas: BigInt!

  syncing: SyncState

  chainID: BigInt!
}
```

Besu 서버로부터 Query로 데이터를 가져올 수 있는 최상위 객체들이 명시되어 있는 부분이다.

- `block(number: Long, hash: Bytes32): Block`
  - 주어진 number 혹은 hash값에 대응되는 block 정보를 반환한다.
  - 인자가 둘 다 주어지지 않을 경우, 가장 최신의 block을 반환한다.
- `blocks(from: Long, to: Long): [Block!]!`
  - from 이상 to 이하의 번호들에 대한 block들을 반환한다.
  - to 값이 없을 경우, 가장 최신의 block 번호가 기본값으로 설정된다.
- `pending: Pending!`
  - 현재의 보류 상태를 반환한다.
- `transaction(hash: Bytes32!): Transaction`
  - hash 값으로 특정되는 트랜잭션을 반환한다.
- `logs(filter: FilterCriteria!): [Log!]!`
  - 주어진 필터 정보와 매칭되는 로그들을 반환한다.
- `gasPrice: BigInt!`
  - 트랜잭션이 채굴되는 것을 보장하는 가스 가격의 추정치를 반환한다.
- `maxPriorityFeePerGas: BigInt!`
  - 트랜잭션이 채굴되는 것을 보장하는 가스 tip 추정치를 반환한다.
- `syncing: SyncState`
  - 현재의 동기화 상태에 대한 정보를 반환한다.
- `chainID: BigInt!`
  - chain ID 값을 반환한다.

현재는 "특정 블록 번호에서의 다중 데이터 조회"를 수행하기 위해 `block` object만을 주로 살펴볼 것이다. 추가적인 기능이 필요 시, 반환되는 Object 정의를 보고 필요한 Object에 대한 argument와 field들을 명시하며 쿼리를 추가 생산해야 한다.

2. Blcok 조회를 위한 필요 스키마

```
type Block {

  # 기타 field 생략

  call(data: CallData!): CallResult
}

input CallData {
  from: Address
  to: Address
  gas: Long
  gasPrice: BigInt
  maxFeePerGas: BigInt
  maxPriorityFeePerGas: BigInt
  value: BigInt
  data: Bytes
}

type CallResult {
  data: Bytes!
  gasUsed: Long!
  status: Long!
}
```

- `Block`
  - 특정 블록 번호에서 스마트 컨트랙트의 데이터를 조회하기 위해, 명시한 블록에서 스마트 컨트랙트의 조회 함수를 실행하기 위해 사용할 Object이다.
  - `call` field에 `CallData` type을 argument로 전달하면, `CallResult` type을 반환한다.
- `CallData`
  - `input` 타입으로 `Block`의 `call` 필드의 인자로만 사용된다.
  - 모든 field들은 선택적(optional)이며, 호출할 대상에 따라 필요한 field들을 정의해야만 정상적으로 호출할 수 있다.
  - 스마트 컨트랙트의 데이터를 조회를 위해서는 컨트랙트 주소가 들어가는 `to` 필드와 실행할 메소드와 파라미터들을 인코딩한 값이 들어가는 `value` 필드를 채워야 한다.
- `CallResult`
  - `data` : 메소드 실행 결과 값이 byte 타입으로 반환된다. (디코딩 필요)
  - `gasUsed` : 메소드 수행에 사용된 gas 양이 반환된다.
  - `status` : 결과 상태 값이 반환되며, 1일 시 성공, 0일 시 실패를 의미한다.

3. 상태 변경을 위한 Mutation

```
type Mutation {
  sendRawTransaction(data: Bytes!): Bytes32!
}
```

- 상태 변경을 일으키는 Mutation Type으로는 `sendRawTransaction` 만이 존재한다.
- `sendRawTransaction` 의 argument로는 **서명된** 트랜잭션이 필요하다.

💡 GraphQL에서 서명 작업은 불가하며, 여러 개의 서명된 트랜잭션을 한 번에 반영시키는 것은 가능하다.

- 결과값으로는 Transaction Hash를 반환한다.

Besu GraphQL 요청 예시

✔ Local 환경에서 Besu의 GraphQL 포트를 8547 (기본값)으로 설정했을 때를 기준으로 terminal 요청 예시입니다.

- 스키마 조회

```
curl -X POST -H "Content-Type: application/json" --data '{ "query": "
{__schema { queryType { fields { name } }}}"}' http://localhost:8547/graphql
```

- 현재 block 기준 call 호출하기

```
curl -X POST -H "Content-Type: application/json" -d'{
"query" : "query getCall($callData: CallData!) {block(number: 15000){call1 :
call(data: $callData){data, gasUsed, status},call2 : call(data: $callData)
{data, status} }}",
"variables" : {"callData" : {"to" :
"0x2114de86c8ea1fd8144c2f1e1e94c74e498afb1b" ,  "data" :
"0x0e37008a0000000000000000000000000000000000000000000000000000000000000001"
}}}' \
http://localhost:8547/graphql
```

- Signed Transaction 날리기

```
curl -X POST -H "Content-Type: application/json" -d'{
"query" : "mutation($data1: Bytes!) {t1: sendRawTransaction(data: $data1)}",
"variables" : { "data1" :
"0xf8a780808302ca51942114de86c8ea1fd8144c2f1e1e94c74e498afb1b80b844fe093d3b00
0000000000000000000000000000000000000000000000000000000000000040000000000000000
0000000000000000000000000000000000000000000000014820f9ea059209daa4c7dab7a31aa
759825aaa646a8ce5947509599839f90b7c26fd2e6d3a04618e838f24590e43167974b0c645af
c0f3ddd682c63906708b2d58a3b283b5a"}
}' \
http://localhost:8547/graphql
```

- 다중 Signed Transaction 날리기

```
curl -X POST -H "Content-Type: application/json" -d'{
"query" : "mutation($data1: Bytes!, $data2: Bytes!) {t1:
sendRawTransaction(data: $data1), t2: sendRawTransaction(data: $data2)}",
"variables" : { "data1" :
"0xf8a780808302ca51942114de86c8ea1fd8144c2f1e1e94c74e498afb1b80b844fe093d3b00
0000000000000000000000000000000000000000000000000000000000004000000000000000
0000000000000000000000000000000000000000000014820f9ea059209daa4c7dab7a31aa
759825aaa646a8ce5947509599839f90b7c26fd2e6d3a04618e838f24590e43167974b0c645af
c0f3ddd682c63906708b2d58a3b283b5a", "data2" :
"0xf8a780808302ca51942114de86c8ea1fd8144c2f1e1e94c74e498afb1b80b844fe093d3b00
0000000000000000000000000000000000000000000000000000000000005000000000000000
0000000000000000000000000000000000000000000014820f9ea0db94708a34f21d396105
889fce9b31b849183aa367ff75664498e5a151c37f2ba064ca58690217e6ce3849b6c5bfa9851
80434eb76bf95e08f722cfea070c7fd9b"}
}' \
http://localhost:8547/graphql
```

Go Module (초안)

1. 특정 블록 번호에서의 스마트 컨트랙트 데이터 다중 조회

```
func BesuCall(bn *big.Int, params ...Call) string {
    q := makeCallMessage(bn, params...)
    rtn := graphqlSend(q)
    return rtn
}

type Call struct {
    To   string `json:"to"`
    Data string `json:"data"`
}
```

- 다중 조회 메소드는 파라미터로 조회 기준 블록 번호와 수행하고자 하는 `Call` 정보들을 받는다.
  - 블록 번호
    - `nil` 혹은 `big.NewInt(0)`으로 입력 시, 가장 최신의 블록을 기준으로 조회한다.
    - `big.NewInt(n)`과 같이 n에 0이 아닌 정수 값 입력 시, 해당 블록 번호 기준으로 조회한다.
  - Call 정보 : `Call` struct
    - `To` : 스마트 컨트랙트 주소
    - `Data` : 실행할 메소드와 파라미터들을 인코딩한 값
- 입력받은 파라미터로 GraphQL 쿼리를 생성한 후, 실행 결과를 String 타입으로 반환한다.
  - Return 값 예시

```
{
```

```
  "data" : {
    "block" : {
      "call1" : {
        "data" :
"0x000000000000000000000000000000000000000000000000000000000000000a",
        "status" : "0x1"
      },
      "call2" : {
        "data" :
"0x000000000000000000000000000000000000000000000000000000000000000a",
        "status" : "0x1"
      }
    }
  }
}
```

2. 다중 서명 트랜잭션 전송

```
func BesuMut(params ...string) string {
    q := makeMutMessage(params...)
    rtn := graphqlSend(q)
    return rtn
}
```

- 다중 서명 트랜잭션 전송은 파라미터로 서명된 값들을 string 타입으로 받는다
- 입력받은 파라미터로 GraphQL 쿼리를 생성한 후, 실행 결과를 String 타입으로 반환한다.
    - Return 값 예시

```
{
  "data" : {
    "t1" :
"0xf2eeda304ce0ed95a9a1a0307676a36969e6c43bbe751ae611c58efca4271332"
  }
}
```

💡 구체적인 GraphQL 활용 방안이 정해지는 것에 따라, 오케스트레이터와 GraphQL 모듈 사이에서 인코딩/디코딩 및 결과값 편집을 수행할 인터페이스가 추가될 예정입니다.

전체 코드

```
package besu_graphql

import (
    "bytes"
    "encoding/json"
```

```
    "fmt"
    "io/ioutil"
    "math/big"
    "net/http"
    "strings"
)

const url string = "http://localhost:8547"

type Call struct {
    To   string `json:"to"`
    Data string `json:"data"`
}

func BesuCall(bn *big.Int, params ...Call) string {
    q := makeCallMessage(bn, params...)
    rtn := graphqlSend(q)
    return rtn
}

func BesuMut(params ...string) string {
    q := makeMutMessage(params...)
    rtn := graphqlSend(q)
    return rtn
}

type besuGraphQL = struct {
    Query    string                 `json:"query"`
    Variable map[string]interface{} `json:"variables"`
}

func newBesuCall(i string, b string, c string, v map[string]interface{})
besuGraphQL { // input, block, call, variable
    return besuGraphQL{
        Query:    fmt.Sprintf("query getCall( %s ) {block%s{ %s }}", i, b, c),
        Variable: v,
    }
}

func newBesuMut(i string, m string, v map[string]interface{}) besuGraphQL {
    return besuGraphQL{
        Query:    fmt.Sprintf("mutation( %s ) {%s}", i, m),
        Variable: v,
    }
}

func graphqlSend(query string) string {

    req, err := http.NewRequest("POST", url, bytes.NewBufferString(query))
    if err != nil {
        panic(err)
    }

    req.Header.Set("Content-Type", "application/json")

    client := &http.Client{}
```

```
    resp, err := client.Do(req)

    if err != nil {
        panic(err)
    }
    defer resp.Body.Close()

    body, _ := ioutil.ReadAll(resp.Body)

    return string(body)

}

func makeCallMessage(bn *big.Int, params ...Call) string {

    var bf string
    if bn == nil || bn.Cmp(big.NewInt(0)) == 0 {
        bf = ""
    } else {
        bf = fmt.Sprintf("(number: %v)", bn)
    }

    // RequestString form
    inf := "$input%d: CallData!"                    // input form
    cf := "call%d : call(data: $%s){data, status}" // call form

    is := make([]string, len(params))  // input slice
    cs := make([]string, len(params))  // call slice
    vm := make(map[string]interface{}) // variable map

    for i, p := range params {
        n := i + 1
        is[i] = fmt.Sprintf(inf, n)
        cs[i] = fmt.Sprintf(cf, n, fmt.Sprintf("input%d", n))
        vm[fmt.Sprintf("input%d", n)] = p
    }

    mc := newBesuCall(strings.Join(is, ", "), bf, strings.Join(cs, ", "), vm)
    r, _ := json.Marshal(mc)
    return string(r)
}

func makeMutMessage(params ...string) string {

    // RequestString form
    inf := "$input%d: Bytes!"                    // input form
    mf := "t%d : sendRawTransaction(data: $%s)" // mutation form

    is := make([]string, len(params))  // input slice
    ms := make([]string, len(params))  // mutation slice
    vm := make(map[string]interface{}) // variable map

    for i, p := range params {
        n := i + 1
        is[i] = fmt.Sprintf(inf, n)
        ms[i] = fmt.Sprintf(mf, n, fmt.Sprintf("input%d", n))
```

```
        vm[fmt.Sprintf("input%d", n)] = p
    }

    mc := newBesuMut(strings.Join(is, ", "), strings.Join(ms, ", "), vm)
    r, _ := json.Marshal(mc)
    return string(r)

}
```