# GraphQL Interpreter

Nazib Sorathiya, Rashmi Chennagiri

## 1 Introduction

GraphQL is a data query and manipulation language for APIs and provides a runtime to fulfil these queries with the already existing data. GraphQL provides a complete and understandable description of the data in the API. It gives the power to the client to get the exact information that is required nothing more than that and nothing less. GraphQL is being used in facebook since 2012 and it's spec was open sourced in 2015 [1].

GraphQL follows a client driven architecture and provides only the information requested by the client in the query. In REST APIs sometimes the unnecessary data is being sent from the server to the client or sometimes the required data is not fulfilled by one API query and requires multiple queries to extract the required information. REST APIs suffer from this problem of over-fetching and under-fetching respectively. GraphQL solves this problem of over-fetching and under-fetching by providing the exact information that is asked in the GraphQL query.

There are several interesting properties of the GraphQL and some of these are listed below [2]:

1. Definite shape of Data: GraphQL mirrors the shape of the queries in its response making it easy to predict the shape of the data in the response and write the query as per their need.
2. Hierarchical: GraphQL follows the hierarchical relationships with the objects which helps in providing all the required information in 1 single query.
3. Strongly Typed: Just like SQL, GraphQL is a strongly typed language where each type describes the fields which helps GraphQL to provide descriptive error messages.
4. Version free: GraphQL provides seemless backward compatibility which discards any need of maintaining different version numbers.

## 2 Motivation

GraphQL is a fairly recent protocol, it have lots of advantages over REST APIs and many companies have started adopting GraphQL. Creating a GraphQL Interpreter as part of the project gives us 2-fold advantage in terms of learning. Firstly, we get the hands-on experience of working with the GraphQL and understand more about its grammar and working. Secondly, gives us an opportunity to bring the course learnings into practice.

We aim to learn multiple things from this project while having fun:

1. Get hands-on experience with the GraphQL
2. Get better understanding of the architecture and the grammar of the GraphQL
3. Implement the course learnings like building AST, interpreting AST in a bigger project to get better understanding

## 3 The GraphQL Interpreter

When a GraphQL server receives a query to process, it generally comes in as a String. This string must be tokenized and parsed into a representation that the machine understands. This representation is called an abstract syntax tree, or AST. When GraphQL processes the query, it walks the tree executing each part against the schema to get the requested data and return in the JSON format.

Figure 1 shows an example of the GraphQL query and data returned by this query in the JSON. It can be seen in this example that JSON response contains only the information about the fields present in the query and follows the same structure.

We will now discuss the main components of the interpreter in more detail.

```
{
  user(id: 4802170) {
    id
    name
    isViewerFriend
    profilePicture(size: 50)  {
      uri
      width
      height
    }
    friendConnection(first: 5) {
      totalCount
      friends {
        id
        name
      }
    }
  }
}
```

```
{
  "data": {
    "user": {
      "id": "4802170",
      "name": "Lee Byron",
      "isViewerFriend": true,
      "profilePicture": {
        "uri": "cdn://pic/4802170/50",
        "width": 50,
        "height": 50
      },
      "friendConnection": {
        "totalCount": 13,
        "friends": [
          {
            "id": "305249",
            "name": "Stephen Schwink"
          },
          {
            "id": "3108935",
            "name": "Nathaniel Roman"
```

**Fig. 1.** GraphQL query example and corresponding response in JSON

### 3.1 Grammar

We have spent a considerable amount of time going through the GraphQL grammar. It has helped us to better understand the language, and also to identify the different parts of the query while processing it. Figure 1 shows a small excerpt of the GraphQL grammar:

*Document* :
    $Definition_{list}$

*Definition* :
    *OperationDefinition*
    *FragmentDefinition*

*OperationDefinition* :
    *SelectionSet*
    *OperationType* $Name_{opt}$ $VariableDefinitions_{opt}$ $Directives_{opt}$ *SelectionSet*

*OperationType* : **one of**
    **query** **mutation** **subscription**

GraphQL grammar excerpt

**Fig. 2.** GraphQL Grammar

The GraphQL grammar is a set of rules which reads from top to bottom [5]. We should be able to infer the following points from the figure 2:

1. A 'Document' is the root of all GraphQL queries. It can consist of one or many Definition's.
2. A 'Definition' is either an 'OperationDefinition' or a 'FragmentDefinition'.
3. An 'OperationDefinition' can either a 'SelectionSet' or 'OperationType'. It can also have additional optional values like 'Name', 'VariableDefinitions', etc.

4. 'OperationType' can only be one of the tokens in [query, mutation, subscription].

The full grammar specification can be found in Appendix B.3 on this page: "http://spec.graphql.org/June2018/#sec-Appendix-Grammar-Summary".

### 3.2 Lexer

We know that lexical analysis is the process of breaking up the stream of input characters into tokens. The GraphQL lexer uses the rules defined in the GraphQL specification to split up the input query into tokens.

We have made use of the Lexer in PLY (Python Lex-Yacc) to build our lexer. Each token contains the following details:

1. Type of token (QUERY, IDENTIFIER, LPAREN, LCURLY, etc.)
2. Actual value of the token
3. Line and column number of the token

Figure 2 shows a sample query which gets broken down into a list of tokens. Since spaces and tabs are supposed to be ignored as part of the GraphQL specification, we have also ensured that none of the white spaces in the original query is processed into a token. [6]

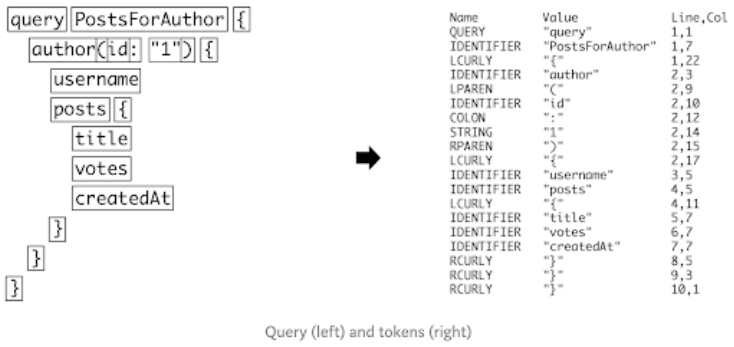


**Fig. 3.** Tokenizing a GraphQL user input query

### 3.3 Parser

We need to turn this list of tokens into an abstract syntax tree (AST). This is where the parser comes into picture. Just like the lexer, the parser also follows the rules defined in the GraphQL specification. Figure 4 shows how the different parts of the GraphQL grammar gets mapped to an actual query.

We have made use of the Yacc parser in PLY (Python Lex-Yacc) to build our parser. It goes through the grammar which we have specified and churns out an AST which our interpreter will use. Figure 5 shows the AST representation that our parser would return for a sample query. [6]

### 3.4 Interpreter

After the parsing stage, we can be sure that the query is syntactically correct, but this does not necessarily imply that it can be successfully executed because there might be errors in the query. The specified field in the query might not exist in the schema, or the datatype of an argument passed might not match with that in the schema. The parser

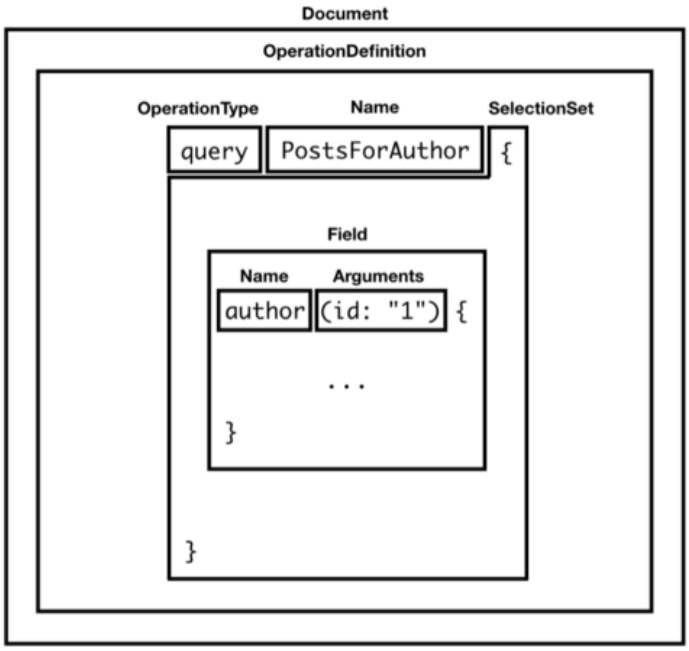


**Fig. 4.** Applying the GraphQL grammar to a query

```
<Document:
  definitions=[
    <Query:
      selections=[
        <Field: name=viewer,
          selections=[
            <Field: name=username>,
            <Field: name=email>
          ]
        >
      ]
    >
  ]
>
```

**Fig. 5.** Output of parser = AST

cannot detect such anomalies, and this is why we now have to perform validation (also according to the GraphQL specification) on the abstract syntax tree. [7]

We have implemented our own interpreter which first validates the AST and then proceeds to interpret it and get the data from the schema. It does all this using the visitor pattern.

Figure 5 shows a visual representation of the AST of an example query. In order to check if the input query can be successfully executed or not, the validate function will traverse the AST using a depth-first traversal. It will start at the root of the AST, i.e. the Document node, and will explore as far as possible along the first branch before backtracking and continuing onto the next branch. In our case, the validation rules are the visitors, which watch as the AST gets traversed. When a particular type of node is reached, its corresponding visitor performs its actions.

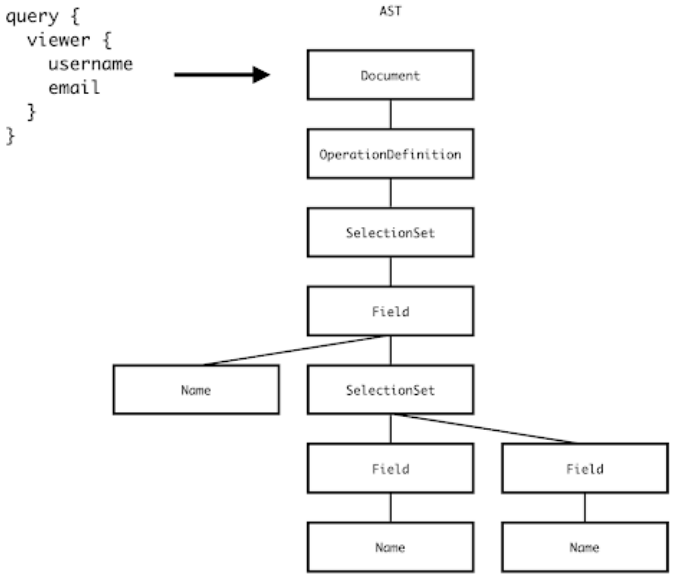


**Fig. 6.** AST representation of a query

While parsing the AST, the specified objects along with the required fields will be fetched from the back-end schema. The response, which includes all the data fetched along with error messages if any, will be returned to the client in the form of a json.

### 3.5 Features Implemented

We have taken an incremental approach to our project. We started by building the interpreter for very basic queries like just asking for a few specific fields on an object. Once we accomplished that, we then added more features such that the interpreter can now handle relatively more complex queries.

As per our experience with the class assignments, we found that it is much easier to implement an interpreter in Python over Java as it eases the datatype checks which makes the tree traversal and recursion easy and takes a lot less lines of code. Hence we decided to use Python for our project even though Java being our primary coding language.

There are several features in the GraphQL language. It is difficult to implement all the features that GraphQL supports. We have implemented a small subset of features some of them are listed below:

1. The most common type of operation that a GraphQL server has to handle is a query. It is denoted using a 'query' keyword, which is optional. The query keyword can also be followed by a query name which is also optional. If the query keyword is not specified, GraphQL interprets the input to be a query by default. If no keyword is specified, and the input format is not like that of a query, the interpreter should notify the end user of the error. Our interpreter can handle all these types of scenarios, all while ignoring comments in them.
2. In GraphQL, you can make a sub-selection of fields for a particular object. Such queries can traverse related objects and their fields, letting clients fetch lots of related data in one request, instead of making several round trips as one would need in a classic REST architecture. Our interpreter can handle basic nested queries. Heavy nesting poses security issues and it should be handled at the business logic level at the client's side. We have implemented 2 levels of nesting.
3. In GraphQL, every field and nested object can get its own set of arguments. This helps to avoid multiple API fetches. Our interpreter can handle any number of arguments passed to the query. When there are no arguments passed in the GraphQL queries, the server should return all the records for that query. Our implemented GraphQL Interpreter allows queries with no arguments, exactly 1 argument, and multiple arguments.

We also added multiple validations to be done on the queries in our GraphQL Interpreter.

1. GraphQL allows the arguments to have various datatypes. Moreover, GraphQL allows the integer type arguments to be either passed as integer or string. In our GraphQL Interpreter we have added this feature that allows the integer type arguments to be passed in integer or string. It should be noted that the same is not allowed for arguments in float type.
2. The GraphQL Interpreter returns an empty list if there is no data with the matching arguments similar to the GraphQL language.
3. If there is more than 1 keyword after the 'query' keyword, then the GraphQL Interpreter should give 'Invalid Syntax' error.
4. When the selection set contains the fields that are not present in actual schema, GraphQL Interpreter should return 'attribute does not exist' error.
5. When the argument that is passed in the query is not a valid field in the schema, the GraphQL Interpreter should return 'invalid argument' error.
6. If the keyword that represent the object in the schema is missing in the query, then the GraphQL Interpreter throws error 'Cannot query field field_name - Object does not exist in database'

We have made use of a mock data file to hold all our schema objects. This made testing easier and faster. We tried to use the testing scripts which we had used as part our homework assignments, but we started facing several dependency issues, and so we had to let that go and spend our efforts to further improve our interpreter, and tested it manually. You can find test evidence at the end of our report.

## 4 Conclusion

We had a lot of fun working on this project. We invested a considerable amount of time learning GraphQL from scratch and how to form complex yet efficient queries using all the features it has to offer, making us appreciate its advantages over REST. We got to see how the back-end servers actually process the input query by learning about the grammar of GraphQL in great detail. Implementing the interpreter made us realise minor but important details of the GraphQL interpreter. We have written around 1200 lines of code, which can also be at https://github.com/nazibs/graphql.

## 5 Future Work

GraphQL supports a lot of features that makes it very convenient to use as a data query and manipulation language and also difficult to implement an interpreter for the language. An extension to this project can be to include various features that graphql supports like Sfragments that lets us create a set of fields and reuse it multiple times, mutations allows us to modify the server side data, variables that allows to pass dynamic arguments, directives that allows to dynamically change the structure of the query, aliases lets us rename the result of a field, etc.

## References


1. https://graphql.org/
2. https://engineering.fb.com/core-data/graphql-a-data-query-language/
3. https://github.com/graphql/graphql-js
4. https://fullstackmark.com/post/17/building-a-graphql-api-with-aspnet-core-2-and-entity-framework-core
5. http://spec.graphql.org/June2018/
6. https://medium.com/@cjoudrey/life-of-a-graphql-query-lexing-parsing-ca7c5045fad8
7. https://medium.com/@cjoudrey/life-of-a-graphql-query-validation-18a8fb52f189
8. https://github.com/ivelum/graphql-py/tree/72baf16d838e82349ee5e8d8f8971ce11cfcedf9
9. https://www.dabeaz.com/ply/ply.html
10. https://my.eng.utah.edu/ cs3100/lectures/l14/ply-3.4/doc/ply.html


## A Test Results

```
(base) Nazibs-MacBook-Pro:graphql nazib$ python interpreter.py
Query:
    query abc {
          droid(id: 2001) {
            # comments
            id
            name
          }
        }

AST for the query:
 <Document: definitions=[<Query: selections=[<Field: name=droid, arguments=[<Argument: name=id, value=2001
>], selections=[<Field: name=id>, <Field: name=name>]>], name=abc>]>

Now interpreting the AST...

Field Name:  droid
Extracted Arguments:  {'id': 2001}
selection_set_fields:  ['id', 'name']

Result:  {"data": {"droid": [{"id": "2001", "name": "R2-D2"}]}}
```

**Fig. 7.** Query Result on Starwar's Droid Data

```
(base) Nazibs-MacBook-Pro:graphql nazib$ python interpreter.py
Query:
    query abc {
          hero(id: 1000) {
            # comments
            id
            name
          }
        }

AST for the query:
 <Document: definitions=[<Query: selections=[<Field: name=hero, arguments=[<Argument: name=id, value=1000>
], selections=[<Field: name=id>, <Field: name=name>]>], name=abc>]>

Now interpreting the AST...

Field Name:  hero
Extracted Arguments:  {'id': 1000}
selection_set_fields:  ['id', 'name']

Result:  {"data": {"hero": [{"id": "1000", "name": "Luke Skywalker"}]}}
(base) Nazibs-MacBook-Pro:graphql nazib$
```

**Fig. 8.** Query Result on Starwar's Hero Data

```
(base) Nazibs-MacBook-Pro:graphql nazib$ python interpreter.py
Query:
    query abc {
          hero(id: 1000) {
            # comments
            id
            name
            home_planet
          }
        }

AST for the query:
 <Document: definitions=[<Query: selections=[<Field: name=hero, arguments=[<Argument: name=id, value=1000>
], selections=[<Field: name=id>, <Field: name=name>, <Field: name=home_planet>]>], name=abc>]>

Now interpreting the AST...

Field Name:  hero
Extracted Arguments:  {'id': 1000}
selection_set_fields:  ['id', 'name', 'home_planet']

Result:  {"data": {"hero": [{"id": "1000", "name": "Luke Skywalker", "home_planet": "Tatooine"}]}}
(base) Nazibs-MacBook-Pro:graphql nazib$
```

**Fig. 9.** Query Result on Starwar's Hero Data

```
(base) Nazibs-MacBook-Pro:graphql nazib$ python interpreter.py
Query:
    query abc {
          droid(id: 2001) {
            # comments
            id
            name
            primary_function
          }
        }

AST for the query:
 <Document: definitions=[<Query: selections=[<Field: name=droid, arguments=[<Argument: name=id, value=2001
>], selections=[<Field: name=id>, <Field: name=name>, <Field: name=primary_function>]>], name=abc>]>

Now interpreting the AST...

Field Name:  droid
Extracted Arguments:  {'id': 2001}
selection_set_fields:  ['id', 'name', 'primary_function']

Result:  {"data": {"droid": [{"id": "2001", "name": "R2-D2", "primary_function": "Astromech"}]}}
(base) Nazibs-MacBook-Pro:graphql nazib$
```

**Fig. 10.** Query Result on Starwar's Droid Data


**Fig. 11.** Query Result without any arguments - listing all records for hero

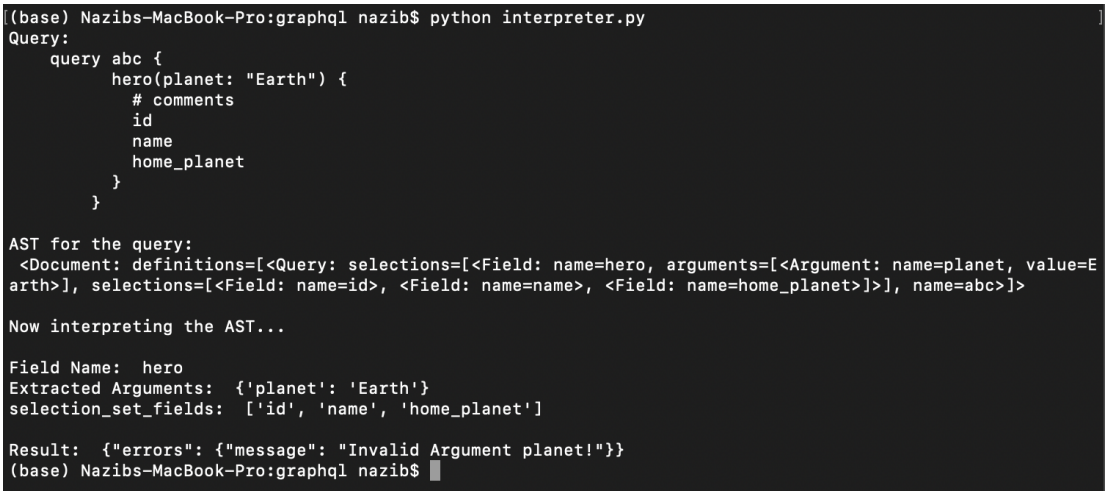


**Fig. 12.** Query Result on passing invalid argument 'planet' instead of correct argument 'home_planet'

```
(base) Nazibs-MacBook-Pro:graphql nazib$ python interpreter.py
Query:
    query abc {
          hero(home_planet: "Tatooine") {
            # comments
            id
            name
            home_planet
          }
        }

AST for the query:
 <Document: definitions=[<Query: selections=[<Field: name=hero, arguments=[<Argument: name=home_planet, va
lue=Tatooine>], selections=[<Field: name=id>, <Field: name=name>, <Field: name=home_planet>]>], name=abc>]
>

Now interpreting the AST...

Field Name:  hero
Extracted Arguments:  {'home_planet': 'Tatooine'}
selection_set_fields:  ['id', 'name', 'home_planet']

Result:  {"data": {"hero": [{"id": "1000", "name": "Luke Skywalker", "home_planet": "Tatooine"}, {"id": "1
001", "name": "Darth Vader", "home_planet": "Tatooine"}]}}
(base) Nazibs-MacBook-Pro:graphql nazib$
```

**Fig. 13.** Query Result on passing valid argument 'home_planet' giving multiple records matching the argument value

```
(base) Nazibs-MacBook-Pro:graphql nazib$ python interpreter.py
Query:
    query abc {
          hero(home_planet: "Tatooine" name: "Darth Vader") {
            # comments
            id
            name
            home_planet
          }
        }

AST for the query:
 <Document: definitions=[<Query: selections=[<Field: name=hero, arguments=[<Argument: name=home_planet, va
lue=Tatooine>, <Argument: name=name, value=Darth Vader>], selections=[<Field: name=id>, <Field: name=name>
, <Field: name=home_planet>]>], name=abc>]>

Now interpreting the AST...

Field Name:  hero
Extracted Arguments:  {'home_planet': 'Tatooine', 'name': 'Darth Vader'}
selection_set_fields:  ['id', 'name', 'home_planet']

Result:  {"data": {"hero": [{"id": "1001", "name": "Darth Vader", "home_planet": "Tatooine"}]}}
(base) Nazibs-MacBook-Pro:graphql nazib$
```

**Fig. 14.** Query Result on Starwar's Hero Data with multiple arguments

```
(base) Nazibs-MacBook-Pro:graphql nazib$ python interpreter.py
Query:
    query abc {
          human(id: 1000) {
            # comments
            id
            name
          }
        }

AST for the query:
 <Document: definitions=[<Query: selections=[<Field: name=human, arguments=[<Argument: name=id, value=1000
>], selections=[<Field: name=id>, <Field: name=name>]>], name=abc>]>

Now interpreting the AST...

Field Name:  human
Extracted Arguments:  {'id': 1000}
selection_set_fields:  ['id', 'name']

Result:  {"errors": {"message": "Cannot query field human - Object does not exist in database"}}
(base) Nazibs-MacBook-Pro:graphql nazib$
```

**Fig. 15.** Query Result on calling invalid object 'human' instead of 'hero'

```
(base) Nazibs-MacBook-Pro:graphql nazib$ python interpreter.py
Query:
    query abc {
          droid(id: 1000) {
            # comments
            id
            name
            primary_function
          }
        }

AST for the query:
 <Document: definitions=[<Query: selections=[<Field: name=droid, arguments=[<Argument: name=id, value=1000
>], selections=[<Field: name=id>, <Field: name=name>, <Field: name=primary_function>]>], name=abc>]>

Now interpreting the AST...

Field Name:  droid
Extracted Arguments:  {'id': 1000}
selection_set_fields:  ['id', 'name', 'primary_function']

Result:  {"data": {"droid": []}}
```

**Fig. 16.** Query Result on Starwar's Droid Data when there was no data for id=1000

```
(base) Nazibs-MacBook-Pro:graphql nazib$ python interpreter.py
Query:
    query abc {
          hero {
            # comments
            id
            name
            primary_function
          }
        }

AST for the query:
 <Document: definitions=[<Query: selections=[<Field: name=hero, selections=[<Field: name=id>, <Field: name
=name>, <Field: name=primary_function>]>], name=abc>]>

Now interpreting the AST...

Field Name:  hero
Extracted Arguments:  {}
selection_set_fields:  ['id', 'name', 'primary_function']

Result:  {"errors": {"message": "Attribute does not exist!"}}
(base) Nazibs-MacBook-Pro:graphql nazib$
```

**Fig. 17.** Query Result on Starwar's Hero Data when invalid attribute 'primary$_function'$ $was passed$

```
(base) Nazibs-MacBook-Pro:graphql nazib$ python interpreter.py
Query:
    query abc {
          hero(home_planet: "Earth") {
            # comments
            id
            name
            home_planet
          }
        }

AST for the query:
 <Document: definitions=[<Query: selections=[<Field: name=hero, arguments=[<Argument: name=home_planet, va
lue=Earth>], selections=[<Field: name=id>, <Field: name=name>, <Field: name=home_planet>]>], name=abc>]>

Now interpreting the AST...

Field Name:  hero
Extracted Arguments:  {'home_planet': 'Earth'}
selection_set_fields:  ['id', 'name', 'home_planet']

Result:  {"data": {"hero": []}}
(base) Nazibs-MacBook-Pro:graphql nazib$
```

**Fig. 18.** Query Result on Starwar's Hero Data when no data existed for the passed argument value

```
(base) Nazibs-MacBook-Pro:graphql nazib$ python interpreter.py
Query:
    query abc {
          hero(home_planet: "Tatooine" name: "Darth Vader") {
            id
            name
            home_planet
          }
        }

AST for the query:
 <Document: definitions=[<Query: selections=[<Field: name=hero, arguments=[<Argument: name=home_planet, va
lue=Tatooine>, <Argument: name=name, value=Darth Vader>], selections=[<Field: name=id>, <Field: name=name>
, <Field: name=home_planet>]>], name=abc>]>

Now interpreting the AST...

Field Name:  hero
Extracted Arguments:  {'home_planet': 'Tatooine', 'name': 'Darth Vader'}
selection_set_fields:  ['id', 'name', 'home_planet']

Result:  {"data": {"hero": [{"id": "1001", "name": "Darth Vader", "home_planet": "Tatooine"}]}}
(base) Nazibs-MacBook-Pro:graphql nazib$
```

**Fig. 19.** Query Result on Starwar's Hero Data without the comment line in query