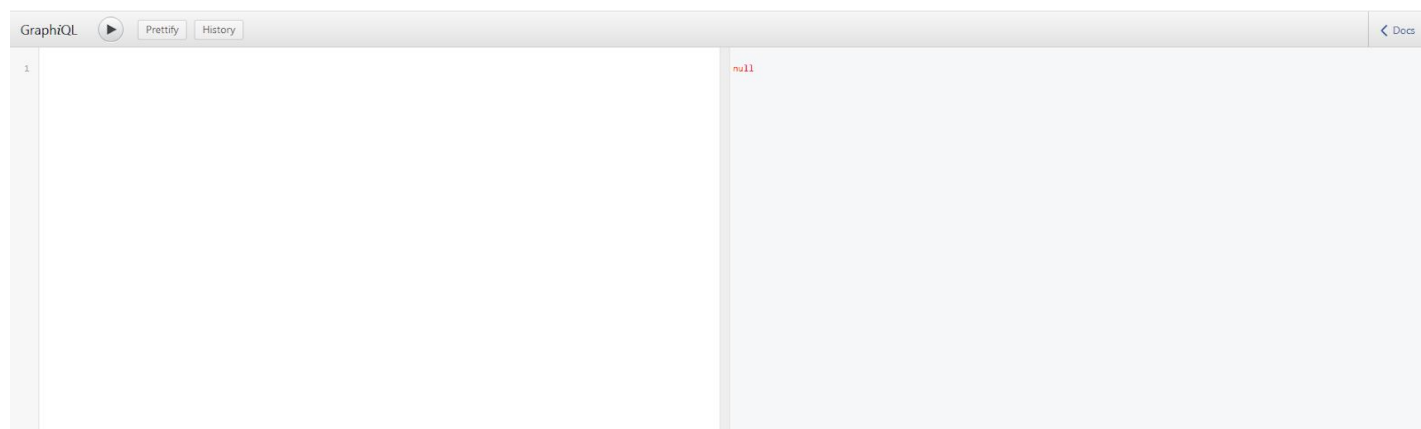


ATTACK
DEFENSE
by PentesterAcademy

Name	GraphQL Basics III
URL	https://attackdefense.com/challengedetails?cid=1994
Type	REST: GraphQL

Important Note: This document illustrates all the important steps required to complete this lab. This is by no means a comprehensive step-by-step solution for this exercise. This is only provided as a reference to various commands needed to complete this exercise and for your further research on this topic. Also, note that the IP addresses and domain names might be different in your lab.

When the lab is launched, the GraphiQL console is provided.



This console could be used to write the GraphQL queries.

Concept 1: Querying data with arguments.

1. Use the following query to fetch the email and the name of the subscribed movies of the person named "John Doe":

Query:

```
{
  searchPerson(name: "John Doe") {
    email
    subscribedMovies {
```

```
edges {  
  node {  
    name  
  }  
}  
}  
}
```

Response:

```
{  
  "data": {  
    "searchPerson": [  
      {  
        "email": "johndoe@example.com",  
        "subscribedMovies": {  
          "edges": [  
            {  
              "node": {  
                "name": "Inception"  
              }  
            },  
            {  
              "node": {  
                "name": "Interstellar"  
              }  
            },  
            {  
              "node": {  
                "name": "The Godfather"  
              }  
            }  
          ]  
        }  
      }  
    ]  
  }  
}
```

2. Use the following query to fetch the name of the persons who have subscribed for the movies named "Inception" and "Rocky":

Query:

```
{
  searchPerson(subscribedMovies: [
    {
      name: "Inception"
    },
    {
      name: "Rocky"
    }
  ]) {
    name
  }
}
```

Response:

```
{
  "data": {
    "searchPerson": [
      {
        "name": "David Smith"
      },
      {
        "name": "Mike Lee"
      }
    ]
  }
}
```

Concept 2: Aliases

Aliases are used to query the same field with different arguments.

Use the following query to fetch the list of subscribed movies for the users named "John Doe" and "David Smith" in a single query.

Query:

```
{
  johnsMovieList: searchPerson(name: "John Doe") {
    subscribedMovies {
```

```

    edges {
      node {
        name
      }
    }
  }
}

davidsMovieList: searchPerson(name: "David Smith") {
  subscribedMovies {
    edges {
      node {
        name
      }
    }
  }
}

```

Response:

```

{
  "data": {
    "johnsMovieList": [
      {
        "subscribedMovies": {
          "edges": [
            {
              "node": {
                "name": "Inception"
              }
            },
            {
              "node": {
                "name": "Interstellar"
              }
            },
            {
              "node": {
                "name": "The Godfather"
              }
            }
          ]
        }
      }
    ]
  }
}

```

```

"daidsMovieList": [
  {
    "subscribedMovies": {
      "edges": [
        {
          "node": {
            "name": "Inception"
          }
        },
        {
          "node": {
            "name": "The Godfather"
          }
        },
        {
          "node": {
            "name": "Venom"
          }
        },
        {
          "node": {
            "name": "Rocky"
          }
        }
      ]
    }
  }
]

```

The response contains the list of favorite movies for both users. This was fetched in a single request to the backend by making use of aliases.

Concept 3: Fragments

Using the previous query as an example, we were fetching the same information for both users. That caused the same query to be written twice. That could be avoided by making use of fragments.

Use the following query to fetch the list of subscribed movies for the users named "John Doe" and "David Smith" in a single query using fragments.

Query:

```

{
  johnsMovieList: searchPerson(name: "John Doe") {

```

```

        ...subscribedMovieList
    }
    davidsMovieList: searchPerson(name: "David Smith") {
        ...subscribedMovieList
    }
}

```

```

fragment subscribedMovieList on Person {
    subscribedMovies {
        edges {
            node {
                name
                rating
            }
        }
    }
}

```

Response:

```

{
  "data": {
    "johnsMovieList": [
      {
        "subscribedMovies": {
          "edges": [
            {
              "node": {
                "name": "Inception",
                "rating": "8.8/10"
              }
            },
            {
              "node": {
                "name": "Interstellar",
                "rating": "8.6/10"
              }
            },
            {
              "node": {
                "name": "The Godfather",
                "rating": "9.2/10"
              }
            }
          ]
        }
      }
    ]
  }
}

```



```

"dauidsMovieList": [
  {
    "subscribedMovies": {
      "edges": [
        {
          "node": {
            "name": "Inception",
            "rating": "8.8/10"
          }
        },
        {
          "node": {
            "name": "The Godfather",
            "rating": "9.2/10"
          }
        },
        {
          "node": {
            "name": "Venom",
            "rating": "6.7/10"
          }
        },
        {
          "node": {
            "name": "Rocky",
            "rating": "8.1/10"
          }
        }
      ]
    }
  }
]
}

```

So, this way, using fragments we were able to remove the duplicate parts in a request.

Note: While typing the queries, the editor would provide suggestions (auto-completion) since the Introspection Queries are not disabled (in default settings).

Concept 4: Operation name

Until now, all the above queries have been using the shorthand syntax and omitting the query keyword and the query name. But naming the queries could be beneficial as it helps in tracking the queries and also very beneficial while debugging as the query name would be logged in the network requests or the GraphQL server logs and would help in tracking down the issues using the query names.

Use the following query to fetch the name and email of all the persons in the database:

Query:

```
query AllPersons {  
  person {  
    edges {  
      node {  
        name  
        email  
      }  
    }  
  }  
}
```

Response:

```
{  
  "data": {  
    "person": {  
      "edges": [  
        {  
          "node": {  
            "name": "John Doe",  
            "email": "johndoe@example.com"  
          }  
        },  
        {  
          "node": {  
            "name": "David Smith",  
            "email": "davidsmith@example.com"  
          }  
        },  
        {  
          "node": {  
            "name": "Michael Jones",  
            "email": "michaeljones@example.com"  
          }  
        }  
      ]  
    }  
  }  
}
```

...

```

{
  "node": {
    "name": "James Garcia",
    "email": "jamesgarcia@example.com"
  }
},
{
  "node": {
    "name": "Maria Gonzalez",
    "email": "mariagonzalez@example.com"
  }
}
]
}
}

```

Here, we have used a named query: AllPersons

Concept 5: Variables

Variables let you write queries with dynamic arguments. That way, we can make multiple requests passing different arguments to retrieve the desired results without having to make any changes to the query at all.

Use the following mutation to add a new movie to the backend database using variables:

Mutation:

```

mutation AddMovie($name: String!, $rating: String!, $releaseYear: Int!) {
  addMovie(name: $name, rating: $rating, releaseYear: $releaseYear) {
    movies {
      name
      rating
      releaseYear
    }
  }
}

```

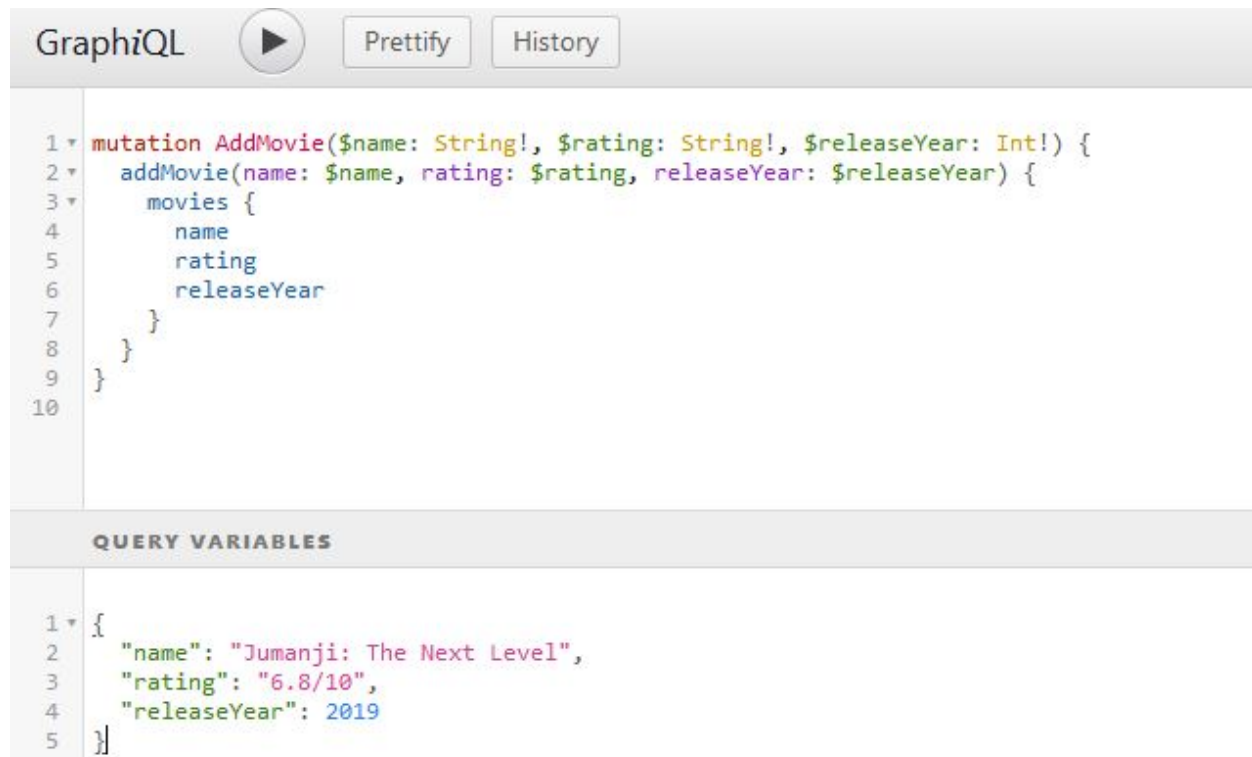
Query Variables:

```

{
  "name": "Jumanji: The Next Level",
  "rating": "6.8/10",

```

```
"releaseYear": 2019
}
```



The screenshot shows the GraphQL IDE interface. At the top, there's a header with the 'GraphiQL' logo, a play button, and buttons for 'Prettify' and 'History'. The main area is divided into two sections. The top section contains a GraphQL mutation query:

```
1 mutation AddMovie($name: String!, $rating: String!, $releaseYear: Int!) {
2   addMovie(name: $name, rating: $rating, releaseYear: $releaseYear) {
3     movies {
4       name
5       rating
6       releaseYear
7     }
8   }
9 }
10
```

 The bottom section, titled 'QUERY VARIABLES', contains the following JSON:

```
1 {
2   "name": "Jumanji: The Next Level",
3   "rating": "6.8/10",
4   "releaseYear": 2019
5 }
```

Note: The exclamation mark in front of the argument type (example: String!) means that the variable is non-nullable. In short, it is required to have some value other.

Response:

```
{
  "data": {
    "addMovie": {
      "movies": {
        "name": "Jumanji: The Next Level",
        "rating": "6.8/10",
        "releaseYear": 2019
      }
    }
  }
}
```

The movie got successfully added to the list of movies.

Notice that just like in queries, if the mutation field returns an object type, we can ask for nested fields. This can be useful for fetching the new state of an object without making multiple requests to the backend.

Now, just by changing the variables, we can add any movie we want utilizing the same mutation.

Concept 6: Directives

Use the following query to fetch the information on a person. The query makes use of variables and directives to control the presence of friends and subscribedMovies fields in the result.

Query:

```
query searchWithCondition($name: String!, $withMovies: Boolean!, $withFriends: Boolean!) {
  searchPerson(name: $name) {
    name
    email
    subscribedMovies @include(if: $withMovies) {
      edges {
        node {
          name
          rating
        }
      }
    }
    friends @include(if: $withFriends) {
      edges {
        node {
          name
        }
      }
    }
  }
}
```

Query Variables:

```
{
  "name": "John Doe",
  "withFriends": false,
  "withMovies": false
}
```

}

Response:

```
{
  "data": {
    "searchPerson": [
      {
        "name": "John Doe",
        "email": "johndoe@example.com"
      }
    ]
  }
}
```

Setting the variable withFriends to true:

Query:

```
1 ▾ query searchWithCondition($name: String!, $withMovies: Boolean!, $withFriends: Boolean!) {
2   searchPerson(name: $name) {
3     name
4     email
5     subscribedMovies @include(if: $withMovies) {
6       edges {
7         node {
8           name
9           rating
10        }
11      }
12    }
13   friends @include(if: $withFriends) {
14     edges {
15       node {
16         name
17       }
18     }
19   }
20 }
21 }
22 }
```

QUERY VARIABLES

```
1 ▾ {
2   "name": "John Doe",
3   "withFriends": true,
4   "withMovies": false
5 }
```

Response:

```
{
  "data": {
    "searchPerson": [
      {
        "name": "John Doe",
        "email": "johndoe@example.com",
        "friends": {
          "edges": [
            {
              "node": {
                "name": "David Smith"
              }
            },
            {
              "node": {
                "name": "Michael Jones"
              }
            }
          ]
        }
      }
    ]
  }
}
```

So, with changing one or more variables, we can control which fields are to be shown in the response and which are to be excluded.

Concept 7: Meta fields

Meta fields contain the metadata about the schema. For instance the `__typename` meta field contains the type information about the name of an object type.

1. Use the following query to get the type name of the object returned:

Query:

```
{
  searchPerson(name: "John Doe") {
    __typename
  }
}
```


Response:

```
{
  "data": {
    "searchPerson": [
      {
        "__typename": "Person"
      }
    ]
  }
}
```

So, the above query returns a Person object.

Similarly, the searchMovies query returns Movies object:

Query:

```
{
  searchMovies(name: "Rocky") {
    __typename
  }
}
```


Response:

```
{
  "data": {
    "searchMovies": [
      {
        "__typename": "Movies"
      }
    ]
  }
}
```

2. Use the following query to identify the type and the fields associated with that entity.

Query:

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

```
fields {  
  name  
  type {  
    name  
  }  
}  
}  
}
```

Response:

```

{
  "data": {
    "_type": {
      "name": "Person",
      "kind": "OBJECT",
      "fields": [
        {
          "name": "id",
          "type": {
            "name": null
          }
        },
        {
          "name": "name",
          "type": {
            "name": "String"
          }
        },
        {
          "name": "email",
          "type": {
            "name": "String"
          }
        },
        {
          "name": "friends",
          "type": {
            "name": "PersonConnection"
          }
        },
        {
          "name": "subscribedMovies",
          "type": {
            "name": "MoviesConnection"
          }
        }
      ]
    }
  }
}

```

Revisiting all the key concepts learnt using the following query:

Use the following query compare 2 persons with respect to their names, email, subscribedMovies and friends.

Query:

```

query comparison($p1: String, $p2: String, $withFriends: Boolean!, $withMovies: Boolean!) {

```

```
p1: searchPerson(name:$p1) {  
  ...comparisonFields  
}  
p2: searchPerson(name:$p2) {  
  ...comparisonFields  
}  
}  
  
fragment comparisonFields on Person {  
  name  
  email  
  subscribedMovies @include(if: $withMovies) {  
    edges {  
      node {  
        name  
        rating  
      }  
    }  
  }  
  friends @include(if: $withFriends) {  
    edges {  
      node {  
        name  
      }  
    }  
  }  
}  
}
```

Query Variables:

```
{  
  "p1": "John Doe",  
  "p2": "David Smith",  
  "withFriends": false,  
  "withMovies": false  
}
```

Response:

```
{
  "data": {
    "p1": [
      {
        "name": "John Doe",
        "email": "johndoe@example.com"
      }
    ],
    "p2": [
      {
        "name": "David Smith",
        "email": "davidsmith@example.com"
      }
    ]
  }
}
```

Since withFriends and withMovies fields were both set to false, only the name and email of the person's having names identified by the variables p1 and p2 got displayed.

Conclusion: In this lab we have dived deeper into GraphQL and learnt many concepts of GraphQL that make it more easy and flexible to work with:

- Querying data with arguments
- Aliases
- Fragments
- Operation name
- Variables
- Directives
- Meta fields

References:

1. GraphQL (<https://graphql.org>)