### Shape, polygon Description automatically generated

### GraphQL is an open-source data query and manipulation language for APIs, and a runtime for fulfilling queries with existing data. GraphQL was developed internally by Facebook in 2012 before being publicly released in 2015. We can use Query in the GraphQL for fetching the data same as the get in the REST API

### Query => GET in REST. Used for fetch data Mutation => PUT, POST, DELETE in REST. Used to modify data. eg(we can write queries and mutations which is used by gitHub) : GitHub GraphQL API [Explorer - GitHub Docs](https://docs.github.com/en/graphql/overview/explorer) sample query query myfirstquery

{

Viewer

{

id

name

location

bio

websiteUrl,

avatarUrl,

login

}

}  
  
**response**  
  
{

"data": {

"viewer": {

"id": "U\_kgDOBlTY3Q",

"name": "Basil Jose",

"location": "Kochi",

"bio": "Over 3 years of experience in software engineering with major skills in C#, ASP.NET MVC, .NET Core, MS SQL Server, Entity Framework, TypeScript, REST API, Graph",

"websiteUrl": "https://www.linkedin.com/in/basil-jose-380a79167/",

"avatarUrl": "https://avatars.githubusercontent.com/u/106223837?v=4",

"login": "ScoluBasil"

}

}

}  
  
  
**Query parameters in GraphQL**  
Every field on a GraphQL object type can have zero or more arguments, for example the length field below:

type Starship {

id: ID!

name: String!

length(unit: LengthUnit = METER): Float

}

All arguments are named. Unlike languages like JavaScript and Python where functions take a list of ordered arguments, all arguments in GraphQL are passed by name specifically. In this case, the length field has one defined argument, unit.

**Arguments** can be either required or optional. When an argument is optional, we can define a default value - if the unit argument is not passed, it will be set to METER by default.  
  
  
  
eg : {

repository(name:"graphQL",owner:"facebook") {

createdAt,

databaseId

}

}

**Output**  
  
{

"data": {

"repository": {

"createdAt": "2015-07-01T01:26:56Z",

"databaseId": 38342221

}

}

}  
  
  
**Schemas in GraphQL**  
  
  
A GraphQL provides a root type for each kind of operations.  
  
Root Types  
-----------------  
  
Query: Queries are used to fetch data.  
Mutations: Mutations are used to add, update and delete a data.  
  
  
Alias in GraphQL  
=====================  
  
we can give the alias to types to distinguish the types  
  
  
eg:   
  
{

dotnetcoreRepository: repository(name: "core", owner: "dotnet") {

createdAt

databaseId

description

descriptionHTML

id

url

}

wpfrepository: repository(name: "wpf", owner: "dotnet") {

createdAt

databaseId

description

descriptionHTML

id

}

}  
  
  
  
response  
==========  
  
{

"data": {

"dotnetcoreRepository": {

"createdAt": "2014-11-18T00:44:57Z",

"databaseId": 26784827,

"description": "Home repository for .NET Core",

"descriptionHTML": "<div>Home repository for .NET Core</div>",

"id": "MDEwOlJlcG9zaXRvcnkyNjc4NDgyNw==",

"url": "https://github.com/dotnet/core"

},

"wpfrepository": {

"createdAt": "2018-10-19T01:55:23Z",

"databaseId": 153711945,

"description": "WPF is a .NET Core UI framework for building Windows desktop applications.",

"descriptionHTML": "<div>WPF is a .NET Core UI framework for building Windows desktop applications.</div>",

"id": "MDEwOlJlcG9zaXRvcnkxNTM3MTE5NDU="

}

}

}  
  
  
**Fragments in GraphQL**  
  
we can create a fragment for the repeated fields in the types   
  
eg:   
  
we can create new fragement like this  
  
fragment RepositoryCommonFiled on Repository

{

createdAt

description

id

}  
  
here the fragment created against the type of repository  
  
and we can use it in our query using … before the fragment name.  
  
  
Eg: {

dotnetcoreRepository: repository (name: "core", owner: "dotnet") {

...RepositoryCommonFiled

url

}

wpfrepository: repository (name: "wpf", owner: "dotnet") {

...RepositoryCommonFiled

}

}

fragment RepositoryCommonFiled on Repository

{

createdAt

description

id

}  
  
  
**response**  
  
{

"data": {

"dotnetcoreRepository": {

"createdAt": "2014-11-18T00:44:57Z",

"description": "Home repository for .NET Core",

"id": "MDEwOlJlcG9zaXRvcnkyNjc4NDgyNw==",

"url": "https://github.com/dotnet/core"

},

"wpfrepository": {

"createdAt": "2018-10-19T01:55:23Z",

"description": "WPF is a .NET Core UI framework for building Windows desktop applications.",

"id": "MDEwOlJlcG9zaXRvcnkxNTM3MTE5NDU="

}

}

}  
  
**Variables in the GraphQL**  
  
in GraphQL variables starts with the $.  
  
  
eg:   
  
query Query Repository($name: String!,$owner: String!)

{

repository(name: $name, owner:$owner)

{

id,

description

}

}  
  
query variables  
  
{

"name": "core”, “owner": "dotnet"

}  
  
**response**  
{

"data": {

"repository": {

"id": "MDEwOlJlcG9zaXRvcnkyNjc4NDgyNw==",

"description": "Home repository for .NET Core"

}

}

}  
  
**Mutations in GraphQL**  
  
mutations are used to add, update, delete   
  
  
Hot Chocolate  
==============  
  
hot chocolate is the package/(or we can say the server for GraphQL) used in .NET core for to use GraphQL.

graphiQL is used now instead of hotChocolate.  
  
install package : GraphQL.Server.transports.AspNetCore.SystemTextJson  
  
is an GraphQL Middleware.  
  
  
GraphiQL : used to write the GraphQL query. Or end point for browser.  
  
  
steps=>

1. Create GraphQL types.

Every type of class needs to inherit from ***ObjectGraphType*** from namespace ‘***GraphQL.Types’***. Which contains all the base classes and properties for GraphQL types.  
  
2. Create GraphQL queries.

We need to create queries for fetch data.  
3. Create GraphQL schemas.

In the schema we can assign our query and mutation. So that graphQL can understand the types for query and mutations.  
4. Register GraphQL types.

We need to register our types, Schema, and GraphQL to service collection in startup.  
5. Create GraphQL mutations.

For modifying the resources, we need to create mutations same as the queries.