graphql-utilities

Feb 22, 2020

Table of Contents:

CHAPTER 1

Introduction

graphql-utilities tries to secure your GraphQL API from malicious queries and provides utilities to make using *graphql-core* easier.

1. It comes with a configurable *ExtendedExecutionContext* class that is capable of performing:
    - **query cost analysis**: define the cost of your queries using the *@cost()* directive provided, *graphql-utilities* provides helper functions and extended execution context to protect you from overly complex or costly queries.
    - **depth limiting**: limit the maximum depth of queries, it's especially useful with object types with recursive relationship
2. It also ships decorators for:
    - **resource-level/one-shot middleware**: middleware in *graphql-core* is run at field-level, it is handly when you need your middleware to run only once, especially auth-related middleware.

CHAPTER 2

Getting Started

You can install graphql-utilities using pip:

```
pip install graphql-utilities
```

Alternatively, if you are using pipenv:

```
pipenv install graphql-utilities
```

Note that **graphql-utilities** requires *graphql-core>=3.0*, which also means you need Python 3.6 or above,

With the package installed, you are ready to go. Depending on your need, you may refer to each page:

- *Using Query Cost Analysis*
- *Limiting Query Depth*
- *Operation-level/One-shot Middleware*

CHAPTER 3

Using Query Cost Analysis

The most exciting part of this library is query cost analysis. It calculates the complexity of queries received and halt execution if the complexity exceeds the permitted value.

Cost analysis works with simple queries, queries with fragments, and mutations.

See also:

For more comprehensive usage, check out *Advanced Usage of Query Cost Analysis*.

3.1 Step 1: Build Schema with Cost Directive

First, you need a custom GraphQL directive: *@cost()* provided by the library.

You can:

1. **import them from 'graphql-utilities'**, then create a schema along with the directive.

```
from graphql import build_schema
from graphql_utilities import cost_directive_source_doc

build_schema(source=cost_directive_source_doc + """
    type Post @cost(complexity: 5) {
        postId: ID!
        title: String
    }
""")
```

2. or **use a helper function 'build_schema_with_cost()'**:

```
from graphql_utilities import build_schema_with_cost

build_schema_with_cost(source="""
    type Post @cost(complexity: 5) {
        postId: ID!
```

(continues on next page)

(continued from previous page)

```
        title: String
    }
""")
```

build_schema_with_cost() is a wrapper of *graphql.build_schema()*. It stitches the **cost directive** with your schema before invoking *build_schema()*. Hence, the function signature of *graphql_utilities.build_schema_with_cost()* is exactly the same as *graphql.build_schema()*:

```
def build_schema_with_cost(
        source: Union[str, Source],
        assume_valid=False,
        assume_valid_sdl=False,
        no_location=False,
        experimental_fragment_variables=False):
```

3.2 Step 2: Define Complexity Of Queries/Objects/Fields (Basic Usage)

To define the complexity of a query, simple add a *@cost()* directive to any **Object** or **Field Definition**.

3.2.1 Complexity of Object

To define the complexity of an Object Type, add *@cost(complexity: <Int>)* directive after the type identifier (*post*, in the case below).

```
type Post @cost(complexity: 5) {
    postId: ID!
    title: String
}

type Query {
    post(postId: ID!): Post
}
```

The complexity of each Post object is **5**. The complexity of querying the schema *post(postId: ID!)* is also **5**.

3.2.2 Complexity of Field

You can always define field-specific complexity by adding a *@cost()* directive to a field:

```
type Post {
    postId: ID!
    title: String
    lastSnapshot: String @cost(complexity: 8)
}

type Query {
    post(postId: ID!): Post
}
```

The complexity of the query below will be **8**.

```
{
    posts(postId: "XXXXXX") {
        postId
        lastSnapshot
    }
}
```

3.2.3 Defining Complexity For Both Object and Fields

In situation when complexity needs to be defined in Object as well as specific Field of the Object, such as:

```
type Post @cost(complexity: 5) {
    postId: ID!
    title: String
    lastSnapshot: String @cost(complexity: 8)
}
```

The total complexity of querying an Object with the field *lastSnapshot* will be the sum of complexities defined in both @cost() directives (*5 + 8 = 13*).

```
{
    posts(postId: "XXXXXX") {
        postId
        lastSnapshot
    }
}
```

The cost of the query above is *5 + 8 = 13*.

3.3 Step 3: Enable Cost Analysis

To enable cost analysis, you must use the *ExtendedExecutionContext* provided by the library.

All you need to do are:

1. Pass *graphql_utilities.ExtendedExecutionContext* as the value of *execution_context_class* into any of *graphql.graphql_sync()*, *graphql.graphql()*, or *graphql.execute()*
2. Pass the following *context_value*:

```
{"cost_analysis": {
   "max_complexity": 5   # Maximum complexity allowed
}}
```

3.3.1 Example:

```
from graphql_utilities import ExtendedExecutionContext, build_schema_with_cost

schema = build_schema_with_cost(source="""
    type Post @cost(complexity: 5) {
        postId: ID!
        title: String   @cost(complexity: 20)
```

(continues on next page)

(continued from previous page)

```
    }

    type Query {
        post(postId: ID!): Post
    }
""")

query = """
    {
        post(postId: "XXXXX") {
            postId
            title
        }
    }
"""

results = graphql_sync(schema=schema,
                       source=query,
                       execution_context_class=ExtendedExecutionContext,
                       context_value={
                           "cost_analysis": {
                               "max_complexity": 8
                           }
                       })

results # ExecutionResult(data=None, errors=[GraphQLError("25 exceeded max complexity␣
↪of 8")])
```

3.4 Advanced Usage

For more advanced usage of query cost analysis, refer to *Advanced Usage of Query Cost Analysis*.

CHAPTER 4

Advanced Usage of Query Cost Analysis

Basic complexity calculation may be insufficient. *graphql-utilities* cost analysis also supports:

- *complexity multiplier*: useful when number of items requested needs to be factored into total complexity
- *overriding Object cost*: useful when the complexity of requesting multiple objects diminishes as the number of items requested increases

Cost analysis works with simple queries, queries with fragments, and mutations.

4.1 Complexity Multiplier - Why and How?

Complexity multiplier allows you to multiply the complexity of Object/Type by the quantity requested.

4.1.1 Why?

For a query that resolves to a list of objects. You probably want to factor the number of items requested into the query complexity.

For instance, you have a *posts* query that returns a list of *Post* objects.

```
type Post @cost(complexity: 5) {
    postId: ID!
    title: String
}

type Query {
    posts(first: Int): [Post]   // The complexity should be `first * 5`
}
```

The complexity should be the value of *first* multiplied by *5*.

4.1.2 How?

All you need to do is to add the *@cost* directive to your *Query* and specify the multipliers as a list of string

```
type Post @cost(complexity: 5) {
    postId: ID!
    title: String
}

type Query {
    posts(first: Int): [Post]   @cost(multiplier: ["first"])
}
```

With the schema above, the complexity of the query below will be *4 * 5 = 20*.

```
{
    posts(first: 4) {
        postId
        title
    }
}
```

4.1.3 What If There Is Field-specific Cost?

If your Object type imposed extra complexities for specific fields as such:

```
type Post @cost(complexity: 5) {
    postId: ID!
    title: String   @cost(complexity: 4)
}

type Query {
    posts(first: Int): [Post]   @cost(multiplier: ["first"])
}
```

and with such query:

```
{
    posts(first: 4) {
        postId
        title
    }
}
```

The complexity will be *4 * (5 + 4) = 36*.

4.2 Overriding Cost of Object Type

You can also override the complexity of Object type defined. All you need is to add the *complexity* argument into the *@cost* directive to the Query type that returns the Object you're overriding.

```
type Post @cost(complexity: 5) {
    postId: ID!
    title: String   @cost(complexity: 4)
```

(continues on next page)

(continued from previous page)

```
}

type Query {
    posts(first: Int!): [Post]   @cost(multiplier: ["first"], complexity: 2)
    post(postId: ID!): Post     // The complexity remains unchanged
}
```

The cost of Post (*5*) will be overridden with *2* when querying *posts(first: Int!): [Post]*.

For instance, the total complexity will be *4 * (2 + 4) = 24*

```
{
    posts(first: 4) {
        postId
        title
    }
}
```

CHAPTER 5

Limiting Query Depth

The *ExtendedExecutionContext* execution context class provided in *graphql-utilities* is capable of limiting the maximum depth of queries, it's especially useful with object types with recursive relationship.

All you need to do are:

1. Pass *graphql_utilities.ExtendedExecutionContext* as the value of *execution_context_class* into any of *graphql.graphql_sync()*, *graphql.graphql()*, or *graphql.execute()*
2. Pass the following *context_value*:

```
{"depth_analysis": {
   "max_depth": 5   # Maximum depth allowed
}}
```

5.1 Example:

```
from graphql_utilities import ExtendedExecutionContext

query = """
    {
        posts(first: 5) {
            postId
            author {
                authorId
                posts(first: 5) {
                    postId
                    author {
                        authorId
                        posts(first: 5) {
                            postId
                            author {
                                authorId
```

(continues on next page)

(continued from previous page)

```
                                posts(first: 5) {
                                    postId
                                    // Depth: 7
                                }
                            }
                        }
                    }
                }
            }
        }
    }
"""

results = graphql_sync(schema=schema,
                       source=query,
                       execution_context_class=ExtendedExecutionContext,
                       context_value={
                           "depth_analysis": {
                               "max_depth": 5
                           }
                       })

results # ExecutionResult(data=None, errors=[GraphQLError("Reached max depth of 5")])
```

CHAPTER 6

Operation-level/One-shot Middleware

The problem of using middleware in *graphql-core* in Python is it run at field-level. If you use a middleware to perform authentication, you probably don't want the authentication logic run when the engine tries to resolve every single field.s

```
from graphql_utilities.decorators import run_only_once

class AuthMiddleware:
    @run_only_once
    def resolve(self, next_, root, info, *args, **kwargs):
        # middleware logic
        return next_(root, info, *args, **kwargs)
```

CHAPTER 7

Introduction

graphql-utilities tries to secure your GraphQL API from malicious queries and provides utilities to make using *graphql-core* easier.

1. It comes with a configurable *ExtendedExecutionContext* class that is capable of performing:
    - **query cost analysis**: define the cost of your queries using the *@cost()* directive provided, *graphql-utilities* provides helper functions and extended execution context to protect you from overly complex or costly queries.
    - **depth limiting**: limit the maximum depth of queries, it's especially useful with object types with recursive relationship
2. It also ships decorators for:
    - **resource-level/one-shot middleware**: middleware in *graphql-core* is run at field-level, it is handly when you need your middleware to run only once, especially auth-related middleware.

CHAPTER 8

Getting Started

You can install graphql-utilities using pip:

```
pip install graphql-utilities
```

Alternatively, if you are using pipenv:

```
pipenv install graphql-utilities
```

Note that **graphql-utilities** requires *graphql-core>=3.0*, which also means you need Python 3.6 or above,

With the package installed, you are ready to go. Depending on your need, you may refer to each page:

- *Using Query Cost Analysis*
- *Limiting Query Depth*
- *Operation-level/One-shot Middleware*