GraphQL-core 3 Documentation

Release 3.0.0b0

Christoph Zwerschke

Sep 14, 2019

Contents

CHAPTER 1

Contents

1.1 Introduction

GraphQL-core-3 is a Python port of GraphQL.js, the JavaScript reference implementation for GraphQL, a query language for APIs created by Facebook.

GraphQL consists of three parts:

- A type system that you define
- A query language that you use to query the API
- An execution and validation engine

The reference implementation closely follows the Specification for GraphQL which consists of the following sections:

- Language
- Type System
- Introspection
- Validation
- Execution
- Response

This division into subsections is reflected in the *Sub-Packages* of GraphQL-core 3. Each of these sub-packages implements the aspects specified in one of the sections of the specification.

1.1.1 Getting started

You can install GraphQL-core 3 using pip:

```
pip install "graphql-core>=3a"
```

You can also install GraphQL-core 3 with pipenv, if you prefer that:

```
pipenv install --pre "graphql-core>=3a"
```

Now you can start using GraphQL-core 3 by importing from the top-level *graphql* package. Nearly everything defined in the sub-packages can also be imported directly from the top-level package.

For instance, using the types defined in the *graphql.type* package, you can define a GraphQL schema, like this simple one:

```
from graphql import (
    GraphQLSchema, GraphQLObjectType, GraphQLField, GraphQLString)

schema = GraphQLSchema(
    query=GraphQLObjectType(
        name='RootQueryType',
        fields={
            'hello': GraphQLField(
                GraphQLString,
                resolve=lambda obj, info: 'world')
        }))
```

The *graphql.execution* package implements the mechanism for executing GraphQL queries. The top-level *graphql()* and *graphql_sync()* functions also parse and validate queries using the *graphql.language* and *graphql.validation* modules.

So to validate and execute a query against our simple schema, you can do:

```
from graphql import graphql_sync

query = '{ hello }'

print(graphql_sync(schema, query))
```

This will yield the following output:

```
ExecutionResult(data={'hello': 'world'}, errors=None)
```

1.1.2 Reporting Issues and Contributing

Please visit the GitHub repository of GraphQL-core 3 if you're interested in the current development or want to report issues or send pull requests.

1.2 Usage

GraphQL-core provides two important capabilities: building a type schema, and serving queries against that type schema.

1.2.1 Building a Type Schema

Using the classes in the *graphql.type* sub-package as building blocks, you can build a complete GraphQL type schema.

Let's take the following schema as an example, which will allow us to query our favorite heroes from the Star Wars trilogy:

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

interface Character {
  id: String!
  name: String
  friends: [Character]
  appearsIn: [Episode]
}

type Human implements Character {
  id: String!
  name: String
  friends: [Character]
  appearsIn: [Episode]
  homePlanet: String
}

type Droid implements Character {
  id: String!
  name: String
  friends: [Character]
  appearsIn: [Episode]
  primaryFunction: String
}

type Query {
  hero(episode: Episode): Character
  human(id: String!): Human
  droid(id: String!): Droid
}
```

We have been using the so called GraphQL schema definition language (SDL) here to describe the schema. While it is also possible to build a schema directly from this notation using GraphQL-core 3, let's first create that schema manually by assembling the types defined here using Python classes, adding resolver functions written in Python for querying the data.

First, we need to import all the building blocks from the *graphql.type* sub-package. Note that you don't need to import from the sub-packages, since nearly everything is also available directly in the top *graphql* package:

```
from graphql import (
    GraphQLArgument, GraphQLEnumType, GraphQLEnumValue,
    GraphQLField, GraphQLInterfaceType, GraphQLList, GraphQLNonNull,
    GraphQLObjectType, GraphQLSchema, GraphQLString)
```

Next, we need to build the enum type `Episode`:

```
episode_enum = GraphQLEnumType('Episode', {
    'NEWHOPE': GraphQLEnumValue(4, description='Released in 1977.'),
    'EMPIRE': GraphQLEnumValue(5, description='Released in 1980.'),
    'JEDI': GraphQLEnumValue(6, description='Released in 1983.')
    }, description='One of the films in the Star Wars Trilogy')
```

If you don't need the descriptions for the enum values, you can also define the enum type like this using an underlying Python `Enum` type:

```
from enum import Enum
```

(continues on next page)

(continued from previous page)

```
class EpisodeEnum(Enum):
    NEWHOPE = 4
    EMPIRE = 5
    JEDI = 6

episode_enum = GraphQLEnumType(
    'Episode', EpisodeEnum,
    description='One of the films in the Star Wars Trilogy')
```

You can also use a Python dictionary instead of a Python `Enum` type to define the GraphQL enum type:

```
episode_enum = GraphQLEnumType(
    'Episode', {'NEWHOPE': 4, 'EMPIRE': 5, 'JEDI': 6},
    description='One of the films in the Star Wars Trilogy')
```

Our schema also contains a `Character` interface. Here is how we build it:

```
character_interface = GraphQLInterfaceType('Character', lambda: {
    'id': GraphQLField(
        GraphQLNonNull(GraphQLString),
        description='The id of the character.'),
    'name': GraphQLField(
        GraphQLString,
        description='The name of the character.'),
    'friends': GraphQLField(
        GraphQLList(character_interface),
        description='The friends of the character,'
                    ' or an empty list if they have none.'),
    'appearsIn': GraphQLField(
        GraphQLList(episode_enum),
        description='Which movies they appear in.'),
    'secretBackstory': GraphQLField(
        GraphQLString,
        description='All secrets about their past.')},
    resolve_type=get_character_type,
    description='A character in the Star Wars Trilogy')
```

Note that we did not pass the dictionary of fields to the `GraphQLInterfaceType` directly, but using a lambda function (a so-called "thunk"). This is necessary because the fields are referring back to the character interface that we are just defining. Whenever you have such recursive definitions in GraphQL-core, you need to use thunks. Otherwise, you can pass everything directly.

Characters in the Star Wars trilogy are either humans or droids. So we define a `Human` and a `Droid` type, which both implement the `Character` interface:

```
human_type = GraphQLObjectType('Human', lambda: {
    'id': GraphQLField(
        GraphQLNonNull(GraphQLString),
        description='The id of the human.'),
    'name': GraphQLField(
        GraphQLString,
        description='The name of the human.'),
    'friends': GraphQLField(
        GraphQLList(character_interface),
        description='The friends of the human,'
                    ' or an empty list if they have none.',
        resolve=get_friends),
```

(continues on next page)

(continued from previous page)

```
    'appearsIn': GraphQLField(
        GraphQLList(episode_enum),
        description='Which movies they appear in.'),
    'homePlanet': GraphQLField(
        GraphQLString,
        description='The home planet of the human, or null if unknown.'),
    'secretBackstory': GraphQLField(
        GraphQLString,
        resolve=get_secret_backstory,
        description='Where are they from'
                    ' and how they came to be who they are.')},
    interfaces=[character_interface],
    description='A humanoid creature in the Star Wars universe.')

droid_type = GraphQLObjectType('Droid', lambda: {
    'id': GraphQLField(
        GraphQLNonNull(GraphQLString),
        description='The id of the droid.'),
    'name': GraphQLField(
        GraphQLString,
        description='The name of the droid.'),
    'friends': GraphQLField(
        GraphQLList(character_interface),
        description='The friends of the droid,'
                    ' or an empty list if they have none.',
        resolve=get_friends,
    ),
    'appearsIn': GraphQLField(
        GraphQLList(episode_enum),
        description='Which movies they appear in.'),
    'secretBackstory': GraphQLField(
        GraphQLString,
        resolve=get_secret_backstory,
        description='Construction date and the name of the designer.'),
    'primaryFunction': GraphQLField(
        GraphQLString,
        description='The primary function of the droid.')
    },
    interfaces=[character_interface],
    description='A mechanical creature in the Star Wars universe.')
```

Now that we have defined all used result types, we can construct the `Query` type for our schema:

```
query_type = GraphQLObjectType('Query', lambda: {
    'hero': GraphQLField(character_interface, args={
        'episode': GraphQLArgument(episode_enum, description=(
            'If omitted, returns the hero of the whole saga.'
            ' If provided, returns the hero of that particular episode.'))},
        resolve=get_hero),
    'human': GraphQLField(human_type, args={
        'id': GraphQLArgument(
            GraphQLNonNull(GraphQLString), description='id of the human')},
        resolve=get_human),
    'droid': GraphQLField(droid_type, args={
        'id': GraphQLArgument(
            GraphQLNonNull(GraphQLString), description='id of the droid')},
        resolve=get_droid)})
```

Using our query type we can define our schema:

```
schema = GraphQLSchema(query_type)
```

Note that you can also pass a mutation type and a subscription type as additional arguments to the `GraphQLSchema`.

1.2.2 Implementing the Resolver Functions

Before we can execute queries against our schema, we also need to define the data (the humans and droids appearing in the Star Wars trilogy) and implement resolver functions that fetch the data (at the beginning of our schema module, because we are referencing them later):

```
luke = dict(
    id='1000', name='Luke Skywalker', homePlanet='Tatooine',
    friends=['1002', '1003', '2000', '2001'], appearsIn=[4, 5, 6])

vader = dict(
    id='1001', name='Darth Vader', homePlanet='Tatooine',
    friends=['1004'], appearsIn=[4, 5, 6])

han = dict(
    id='1002', name='Han Solo', homePlanet=None,
    friends=['1000', '1003', '2001'], appearsIn=[4, 5, 6])

leia = dict(
    id='1003', name='Leia Organa', homePlanet='Alderaan',
    friends=['1000', '1002', '2000', '2001'], appearsIn=[4, 5, 6])

tarkin = dict(
    id='1004', name='Wilhuff Tarkin', homePlanet=None,
    friends=['1001'], appearsIn=[4])

human_data = {
    '1000': luke, '1001': vader, '1002': han, '1003': leia, '1004': tarkin}

threepio = dict(
    id='2000', name='C-3PO', primaryFunction='Protocol',
    friends=['1000', '1002', '1003', '2001'], appearsIn=[4, 5, 6])

artoo = dict(
    id='2001', name='R2-D2', primaryFunction='Astromech',
    friends=['1000', '1002', '1003'], appearsIn=[4, 5, 6])

droid_data = {
    '2000': threepio, '2001': artoo}


def get_character_type(character, _info, _type):
        return 'Droid' if character['id'] in droid_data else 'Human'


def get_character(id):
    """Helper function to get a character by ID."""
    return human_data.get(id) or droid_data.get(id)


def get_friends(character, _info):
```

(continues on next page)

(continued from previous page)

```
    """Allows us to query for a character's friends."""
    return map(get_character, character.friends)


def get_hero(root, _info, episode):
    """Allows us to fetch the undisputed hero of the trilogy, R2-D2."""
    if episode == 5:
        return luke  # Luke is the hero of Episode V
    return artoo  # Artoo is the hero otherwise


def get_human(root, _info, id):
    """Allows us to query for the human with the given id."""
    return human_data.get(id)


def get_droid(root, _info, id):
    """Allows us to query for the droid with the given id."""
    return droid_data.get(id)


def get_secret_backstory(_character, _info):
    """Raise an error when attempting to get the secret backstory."""
    raise RuntimeError('secretBackstory is secret.')
```

Note that the resolver functions get the current object as first argument. For a field on the root Query type this is often not used, but a root object can also be defined when executing the query. As the second argument, they get an object containing execution information, as defined in the *graphql.type.GraphQLResolveInfo* class. This object also has a `context` attribute that can be used to provide every resolver with contextual information like the currently logged in user, or a database session. In our simple example we don't authenticate users and use static data instead of a database, so we don't make use of it here. In addition to these two arguments, resolver functions optionally get the defined for the field in the schema, using the same names (the names are not translated from GraphQL naming conventions to Python naming conventions).

Also note that you don't need to provide resolvers for simple attribute access or for fetching items from Python dictionaries.

Finally, note that our data uses the internal values of the `Episode` enum that we have defined above, not the descriptive enum names that are used externally. For example, `NEWHOPE` ("A New Hope") has internally the actual episode number 4 as value.

1.2.3 Executing Queries

Now that we have defined the schema and breathed life into it with our resolver functions, we can execute arbitrary query against the schema.

The *graphql* package provides the *graphql.graphql()* function to execute queries. This is the main feature of GraphQL-core.

Note however that this function is actually a coroutine intended to be used in asynchronous code running in an event loop.

Here is one way to use it:

```
import asyncio
from graphql import graphql
```

(continues on next page)

(continued from previous page)

```
async def query_artoo():
    result = await graphql(schema, """
    {
      droid(id: "2001") {
        name
        primaryFunction
      }
    }
    """)
    print(result)

asyncio.get_event_loop().run_until_complete(query_artoo())
```

In our query, we asked for the droid with the id 2001, which is R2-D2, and its primary function, Astromech. When everything has been implemented correctly as shown above, you should get the expected result:

```
ExecutionResult(
    data={'droid': {'name': 'R2-D2', 'primaryFunction': 'Astromech'}},
    errors=None)
```

The *ExecutionResult* has a `data` attribute with the actual result, and an `errors` attribute with a list of errors if there were any.

If all your resolvers work synchronously, as in our case, you can also use the *graphql.graphql_sync()* function to query the result in ordinary synchronous code:

```
from graphql import graphql_sync

result = graphql_sync(schema, """
  query FetchHuman($id: String!) {
    human(id: $id) {
      name
      homePlanet
    }
  }
  """, variable_values={'id': '1000'})
print(result)
```

Here we asked for the human with the id 1000, Luke Skywalker, and his home planet, Tatooine. So the output of the code above is:

```
ExecutionResult(
    data={'human': {'name': 'Luke Skywalker', 'homePlanet': 'Tatooine'}},
    errors=None)
```

Let's see what happens when we make a mistake in the query, by querying a non-existing `homeTown` field:

```
result = graphql_sync(schema, """
    {
      human(id: "1000") {
        name
        homeTown
      }
    }
    """)
print(result)
```

You will get the following result as output:

```
ExecutionResult(data=None, errors=[GraphQLError(
    "Cannot query field 'homeTown' on type 'Human'. Did you mean 'homePlanet'?",
    locations=[SourceLocation(line=5, column=9)])])
```

This is very helpful. Not only do we get the exact location of the mistake in the query, but also a suggestion for correcting the bad field name.

GraphQL also allows to request the meta field `__typename`. We can use this to verify that the hero of "The Empire Strikes Back" episode is Luke Skywalker and that he is in fact a human:

```
result = graphql_sync(schema, """
    {
      hero(episode: EMPIRE) {
        __typename
        name
      }
    }
    """)
print(result)
```

This gives the following output:

```
ExecutionResult(
    data={'hero': {'__typename': 'Human', 'name': 'Luke Skywalker'}},
    errors=None)
```

Finally, let's see what happens when we try to access the secret backstory of our hero:

```
result = graphql_sync(schema, """
    {
      hero(episode: EMPIRE) {
        name
        secretBackstory
      }
    }
    """)
print(result)
```

While we get the name of the hero, the secret backstory fields remains empty, since its resolver function raises an error. However, we get the error that has been raised by the resolver in the `errors` attribute of the result:

```
ExecutionResult(
    data={'hero': {'name': 'Luke Skywalker', 'secretBackstory': None}},
    errors=[GraphQLError('secretBackstory is secret.',
            locations=[SourceLocation(line=5, column=9)],
            path=['hero', 'secretBackstory'])])
```

1.2.4 Using the Schema Definition Language

Above we defined the GraphQL schema as Python code, using the `GraphQLSchema` class and other classes representing the various GraphQL types.

GraphQL-core 3 also provides a language-agnostic way of defining a GraphQL schema using the GraphQL schema definition language (SDL) which is also part of the GraphQL specification. To do this, we simply feed the SDL as a string to the *`graphql.utilities.build_schema()`* function:

```
from graphql import build_schema

schema = build_schema("""

    enum Episode { NEWHOPE, EMPIRE, JEDI }

    interface Character {
      id: String!
      name: String
      friends: [Character]
      appearsIn: [Episode]
    }

    type Human implements Character {
      id: String!
      name: String
      friends: [Character]
      appearsIn: [Episode]
      homePlanet: String
    }

    type Droid implements Character {
      id: String!
      name: String
      friends: [Character]
      appearsIn: [Episode]
      primaryFunction: String
    }

    type Query {
      hero(episode: Episode): Character
      human(id: String!): Human
      droid(id: String!): Droid
    }
    """)
```

The result is a `GraphQLSchema` object just like the one we defined above, except for the resolver functions which cannot be defined in the SDL.

We would need to manually attach these functions to the schema, like so:

```
schema.query_type.fields['hero'].resolve = get_hero
schema.get_type('Character').resolve_type = get_character_type
```

Another problem is that the SDL does not define the server side values of the `Episode` enum type which are returned by the resolver functions and which are different from the names used for the episode.

So we would also need to manually define these values, like so:

```
for name, value in schema.get_type('Episode').values.items():
    value.value = EpisodeEnum[name].value
```

This would allow us to query the schema built from SDL just like the manually assembled schema:

```
from graphql import graphql_sync

result = graphql_sync(schema, """
    {
```

(continues on next page)

(continued from previous page)

```
      hero(episode: EMPIRE) {
        name
        appearsIn
      }
    }
    """)
print(result)
```

And we would get the expected result:

```
ExecutionResult(
    data={'hero': {'name': 'Luke Skywalker',
                   'appearsIn': ['NEWHOPE', 'EMPIRE', 'JEDI']}},
    errors=None)
```

1.2.5 Using resolver methods

Above we have attached resolver functions to the schema only. However, it is also possible to define resolver methods on the resolved objects, starting with the `root_value` object that you can pass to the *graphql.graphql()* function when executing a query.

In our case, we could create a `Root` class with three methods as root resolvers, like so:

```
class Root:
    """The root resolvers"""

    def hero(self, info, episode):
        return luke if episode == 5 else artoo

    def human(self, info, id):
        return human_data.get(id)

    def droid(self, info, id):
        return droid_data.get(id)
```

Since we have defined synchronous methods only, we will use the *graphql.graphql_sync()* function to execute a query, passing a `Root()` object as the `root_value`:

```
from graphql import graphql_sync

result = graphql_sync(schema, """
    {
      droid(id: "2001") {
        name
        primaryFunction
      }
    }
    """, Root())
print(result)
```

Even if we haven't attached a resolver to the `hero` field as we did above, this would now still resolve and give the following output:

```
ExecutionResult(
    data={'droid': {'name': 'R2-D2', 'primaryFunction': 'Astromech'}},
    errors=None)
```

Of course you can also define asynchronous methods as resolvers, and execute queries asynchronously with *graphql.graphql()*.

In a similar vein, you can also attach resolvers as methods to the resolved objects on deeper levels than the root of the query. In that case, instead of resolving to dictionaries with keys for all the fields, as we did above, you would resolve to objects with attributes for all the fields. For instance, you would define a class `Human` with a method `friends()` for resolving the friends of a human. You can also make use of inheritance in this case. The `Human` class and a `Droid` class could inherit from a `Character` class and use its methods as resolvers for common fields.

1.2.6 Using an Introspection Query

A third way of building a schema is using an introspection query on an existing server. This is what GraphiQL uses to get information about the schema on the remote server. You can create an introspection query using GraphQL-core 3 with:

```
from graphql import get_introspection_query

query = get_introspection_query(descriptions=True)
```

This will also yield the descriptions of the introspected schema fields. You can also create a query that omits the descriptions with:

```
query = get_introspection_query(descriptions=False)
```

In practice you would run this query against a remote server, but we can also run it against the schema we have just built above:

```
from graphql import graphql_sync

introspection_query_result = graphql_sync(schema, query)
```

The `data` attribute of the introspection query result now gives us a dictionary, which constitutes a third way of describing a GraphQL schema:

```
{'__schema': {
    'queryType': {'name': 'Query'},
    'mutationType': None, 'subscriptionType': None,
    'types': [
        {'kind': 'OBJECT', 'name': 'Query', 'description': None,
         'fields': [{
            'name': 'hero', 'description': None,
            'args': [{'name': 'episode', 'description': ... }],
            ... }, ... ], ... },
        ... ],
    ... }
}
```

This result contains all the information that is available in the SDL description of the schema, i.e. it does not contain the resolve functions and information on the server-side values of the enum types.

You can convert the introspection result into `GraphQLSchema` with GraphQL-core 3 by using the *graphql.utilities.build_client_schema()* function:

```
from graphql import build_client_schema

client_schema = build_client_schema(introspection_query_result.data)
```

It is also possible to convert the result to SDL with GraphQL-core 3 by using the *graphql.utilities.print_schema()* function:

```
from graphql import print_schema

sdl = print_schema(client_schema)
print(sdl)
```

This prints the SDL representation of the schema that we started with.

As you see, it is easy to convert between the three forms of representing a GraphQL schema in GraphQL-core 3.

1.2.7 Parsing GraphQL Queries and Schema Notation

When executing GraphQL queries, the first step that happens under the hood is parsing the query. But GraphQL-core 3 also exposes the parser for direct usage via the *graphql.language.parse()* function. When you pass this function a GraphQL source code, it will be parsed and returned as a Document, i.e. an abstract syntax tree (AST) of *graphql.language.Node* objects. The root node will be a *graphql.language.DocumentNode*, with child nodes of different kinds corresponding to the GraphQL source. The nodes also carry information on the location in the source code that they correspond to.

Here is an example:

```
from graphql import parse

document = parse("""
    type Query {
      me: User
    }

    type User {
      id: ID
      name: String
    }
""")
```

You can also leave out the information on the location in the source code when creating the AST document:

```
document = parse(..., no_location=True)
```

This will give the same result as manually creating the AST document:

```
from graphql.language.ast import *

document = DocumentNode(definitions=[
    ObjectTypeDefinitionNode(
        name=NameNode(value='Query'),
        fields=[
            FieldDefinitionNode(
                name=NameNode(value='me'),
                type=NamedTypeNode(name=NameNode(value='User')),
                arguments=[], directives=[])
```

(continues on next page)

(continued from previous page)

```
            ], directives=[], interfaces=[]),
    ObjectTypeDefinitionNode(
        name=NameNode(value='User'),
        fields=[
            FieldDefinitionNode(
                name=NameNode(value='id'),
                type=NamedTypeNode(
                    name=NameNode(value='ID')),
                arguments=[], directives=[]),
            FieldDefinitionNode(
                name=NameNode(value='name'),
                type=NamedTypeNode(
                    name=NameNode(value='String')),
                arguments=[], directives=[]),
            ], directives=[], interfaces=[]),
    ])
```

When parsing with `no_location=False` (the default), the AST nodes will also have a `loc` attribute carrying the information on the source code location corresponding to the AST nodes.

When there is a syntax error in the GraphQL source code, then the *`parse()`* function will raise a *`graphql.error.GraphQLSyntaxError`*.

The parser can not only be used to parse GraphQL queries, but also to parse the GraphQL schema definition language. This will result in another way of representing a GraphQL schema, as an AST document.

1.2.8 Extending a Schema

With GraphQL-core 3 you can also extend a given schema using type extensions. For example, we might want to add a `lastName` property to our `Human` data type to retrieve only the last name of the person.

This can be achieved with the *`graphql.utilities.extend_schema()`* function as follows:

```
from graphql import extend_schema, parse

schema = extend_schema(schema, parse("""
    extend type Human {
     lastName: String
    }
    """))
```

Note that this function expects the extensions as an AST, which we can get using the *`graphql.language.parse()`* function. Also note that the *extend_schema* function does not alter the original schema, but returns a new schema object.

We also need to attach a resolver function to the new field:

```
def get_last_name(human, info):
    return human['name'].rsplit(None, 1)[-1]

schema.get_type('Human').fields['lastName'].resolve = get_last_name
```

Now we can query only the last name of a human:

```
from graphql import graphql_sync
```

(continues on next page)

(continued from previous page)

```
result = graphql_sync(schema, """
    {
      human(id: "1000") {
        lastName
        homePlanet
      }
    }
    """)
print(result)
```

This query will give the following result:

```
ExecutionResult(
    data={'human': {'lastName': 'Skywalker', 'homePlanet': 'Tatooine'}},
    errors=None)
```

1.2.9 Validating GraphQL Queries

When executing GraphQL queries, the second step that happens under the hood after parsing the source code is a validation against the given schema using the rules of the GraphQL specification. You can also run the validation step manually by calling the *graphql.validation.validate()* function, passing the schema and the AST document:

```
from graphql import parse, validate

errors = validate(schema, parse("""
    {
      human(id: NEWHOPE) {
        name
        homeTown
        friends
      }
    }
    """))
```

As a result, you will get a complete list of all errors that the validators has found. In this case, we will get:

```
[GraphQLError(
    'Expected type String!, found NEWHOPE.',
    locations=[SourceLocation(line=3, column=17)]),
 GraphQLError(
    "Cannot query field 'homeTown' on type 'Human'."
     " Did you mean 'homePlanet'?",
     locations=[SourceLocation(line=5, column=9)]),
 GraphQLError(
    "Field 'friends' of type '[Character]' must have a sub selection of subfields."
     "  Did you mean 'friends { ... }'?",
     locations=[SourceLocation(line=6, column=9)])]
```

These rules are available in the *specified_rules* list and implemented in the `graphql.validation.rules` subpackage. Instead of the default rules, you can also use a subset or create custom rules. The rules are based on the *graphql.validation.ValidationRule* class which is based on the *graphql.language.Visitor* class which provides a way of walking through an AST document using the visitor pattern.

1.2.10 Subscriptions

Sometimes you need to not only query data from a server, but you also want to push data from the server to the client. GraphQL-core 3 has you also covered here, because it implements the "Subscribe" algorithm described in the GraphQL spec. To execute a GraphQL subscription, you must use the *subscribe()* method from the *graphql.subscription* module. Instead of a single `ExecutionResult`, this function returns an asynchronous iterator yielding a stream of those, unless there was an immediate error. Of course you will then also need to maintain a persistent channel to the client (often realized via WebSockets) to push these results back.

1.2.11 Other Usages

GraphQL-core 3 provides many more low-level functions that can be used to work with GraphQL schemas and queries. We encourage you to explore the contents of the various *Sub-Packages*, particularly *graphql.utilities*, and to look into the source code and tests of GraphQL-core 3 in order to find all the functionality that is provided and understand it in detail.

1.3 Reference

GraphQL-core

The primary *graphql* package includes everything you need to define a GraphQL schema and fulfill GraphQL requests.

GraphQL-core provides a reference implementation for the GraphQL specification but is also a useful utility for operating on GraphQL files and building sophisticated tools.

This top-level package exports a general purpose function for fulfilling all steps of the GraphQL specification in a single operation, but also includes utilities for every part of the GraphQL specification:

- Parsing the GraphQL language.
- Building a GraphQL type schema.
- Validating a GraphQL request against a type schema.
- Executing a GraphQL request against a type schema.

This also includes utility functions for operating on GraphQL types and GraphQL documents to facilitate building tools.

You may also import from each sub-package directly. For example, the following two import statements are equivalent:

```
from graphql import parse
from graphql.language import parse
```

The sub-packages of GraphQL-core 3 are:

- *graphql.language*: Parse and operate on the GraphQL language.
- *graphql.type*: Define GraphQL types and schema.
- *graphql.validation*: The Validation phase of fulfilling a GraphQL result.
- *graphql.execution*: The Execution phase of fulfilling a GraphQL request.
- *graphql.error*: Creating and formatting GraphQL errors.
- ***graphql.utilities*:** Common useful computations upon the GraphQL language and type objects.
- *graphql.subscription*: Subscribe to data updates.

1.3.1 Top-Level Functions

graphql.**graphql**(*schema: graphql.type.schema.GraphQLSchema, source: Union[str, graphql.language.source.Source], root_value: Any = None, context_value: Any = None, variable_values: Dict[str, Any] = None, operation_name: str = None, field_resolver: Callable[[...], Any] = None, type_resolver: Callable[[Any, graphql.type.definition.GraphQLResolveInfo, GraphQLAbstractType], Union[Awaitable[Union[GraphQLObjectType, str, None]], GraphQLObjectType, str, None]] = None, middleware: Union[Tuple, List[T], graphql.execution.middleware.MiddlewareManager, None] = None, execution_context_class: Type[graphql.execution.execute.ExecutionContext] = <class 'graphql.execution.execute.ExecutionContext'>*) → graphql.execution.execute.ExecutionResult

Execute a GraphQL operation asynchronously.

This is the primary entry point function for fulfilling GraphQL operations by parsing, validating, and executing a GraphQL document along side a GraphQL schema.

More sophisticated GraphQL servers, such as those which persist queries, may wish to separate the validation and execution phases to a static time tooling step, and a server runtime step.

Accepts the following arguments:

Parameters

- **schema** – The GraphQL type system to use when validating and executing a query.
- **source** – A GraphQL language formatted string representing the requested operation.
- **root_value** – The value provided as the first argument to resolver functions on the top level type (e.g. the query object type).
- **context_value** – The context value is provided as an attribute of the second argument (the resolve info) to resolver functions. It is used to pass shared information useful at any point during query execution, for example the currently logged in user and connections to databases or other services.
- **variable_values** – A mapping of variable name to runtime value to use for all variables defined in the request string.
- **operation_name** – The name of the operation to use if request string contains multiple possible operations. Can be omitted if request string contains only one operation.
- **field_resolver** – A resolver function to use when one is not provided by the schema. If not provided, the default field resolver is used (which looks for a value or method on the source value with the field's name).
- **type_resolver** – A type resolver function to use when none is provided by the schema. If not provided, the default type resolver is used (which looks for a *__typename* field or alternatively calls the *is_type_of* method).
- **middleware** – The middleware to wrap the resolvers with
- **execution_context_class** – The execution context class to use to build the context

graphql.**graphql_sync**(*schema: graphql.type.schema.GraphQLSchema, source: Union[str, graphql.language.source.Source], root_value: Any = None, context_value: Any = None, variable_values: Dict[str, Any] = None, operation_name: str = None, field_resolver: Callable[[...], Any] = None, type_resolver: Callable[[Any, graphql.type.definition.GraphQLResolveInfo, GraphQLAbstractType], Union[Awaitable[Union[GraphQLObjectType, str, None]], GraphQLObjectType, str, None]] = None, middleware: Union[Tuple, List[T], graphql.execution.middleware.MiddlewareManager, None] = None, execution_context_class: Type[graphql.execution.execute.ExecutionContext] = <class 'graphql.execution.execute.ExecutionContext'>*) → graphql.execution.execute.ExecutionResult

Execute a GraphQL operation synchronously.

The graphql_sync function also fulfills GraphQL operations by parsing, validating, and executing a GraphQL document along side a GraphQL schema. However, it guarantees to complete synchronously (or throw an error) assuming that all field resolvers are also synchronous.

1.3.2 Sub-Packages

Error

GraphQL Errors

The *graphql.error* package is responsible for creating and formatting GraphQL errors.

exception graphql.error.**GraphQLError**(*message: str, nodes: Union[Sequence[Node], Node] = None, source: Source = None, positions: Sequence[int] = None, path: Sequence[Union[str, int]] = None, original_error: Exception = None, extensions: Dict[str, Any] = None*)

GraphQL Error

A GraphQLError describes an Error found during the parse, validate, or execute phases of performing a GraphQL operation. In addition to a message, it also includes information about the locations in a GraphQL document and/or execution result that correspond to the Error.

extensions
Extension fields to add to the formatted error

formatted
Get error formatted according to the specification.

locations
Source locations

A list of (line, column) locations within the source GraphQL document which correspond to this error.

Errors during validation often contain multiple locations, for example to point out two things with the same name. Errors during execution include a single location, the field which produced the error.

message
A message describing the Error for debugging purposes

Note: should be treated as readonly, despite invariant usage.

nodes
A list of GraphQL AST Nodes corresponding to this error

original_error
The original error thrown from a field resolver during execution

path
A list of field names and array indexes describing the JSON-path into the execution response which corresponds to this error.

Only included for errors during execution.

positions
Error positions

A list of character offsets within the source GraphQL document which correspond to this error.

source
The source GraphQL document for the first location of this error

Note that if this Error represents more than one node, the source may not represent nodes after the first node.

exception graphql.error.**GraphQLSyntaxError**(*source*, *position*, *description*)
A GraphQLError representing a syntax error.

graphql.error.**format_error**(*error: GraphQLError*) → Dict[str, Any]
Format a GraphQL error.

Given a GraphQLError, format it according to the rules described by the "Response Format, Errors" section of the GraphQL Specification.

graphql.error.**located_error**(*original_error: Union[Exception, graphql.error.graphql_error.GraphQLError], nodes: Sequence[Node], path: Sequence[Union[str, int]]*) → graphql.error.graphql_error.GraphQLError
Located GraphQL Error

Given an arbitrary Error, presumably thrown while attempting to execute a GraphQL operation, produce a new GraphQLError aware of the location in the document responsible for the original Error.

graphql.error.**print_error**(*error: graphql.error.graphql_error.GraphQLError*) → str
Print a GraphQLError to a string.

Represents useful location information about the error's position in the source.

graphql.error.**INVALID = <INVALID>**
Symbol for invalid or undefined values

This singleton object is used to describe invalid or undefined values. It corresponds to the `undefined` value in GraphQL.js.

Execution

GraphQL Execution

The *graphql.execution* package is responsible for the execution phase of fulfilling a GraphQL request.

graphql.execution.**execute**(*schema: graphql.type.schema.GraphQLSchema, document: graphql.language.ast.DocumentNode, root_value: Any = None, context_value: Any = None, variable_values: Dict[str, Any] = None, operation_name: str = None, field_resolver: Callable[[...], Any] = None, type_resolver: Callable[[Any, graphql.type.definition.GraphQLResolveInfo, GraphQLAbstractType], Union[Awaitable[Union[GraphQLObjectType, str, None]], GraphQLObjectType, str, None]] = None, middleware: Union[Tuple, List[T], graphql.execution.middleware.MiddlewareManager, None] = None, execution_context_class: Type[ExecutionContext] = None*) → Union[Awaitable[graphql.execution.execute.ExecutionResult], graphql.execution.execute.ExecutionResult]

Execute a GraphQL operation.

Implements the "Evaluating requests" section of the GraphQL specification.

Returns an ExecutionResult (if all encountered resolvers are synchronous), or a coroutine object eventually yielding an ExecutionResult.

If the arguments to this function do not result in a legal execution context, a GraphQLError will be thrown immediately explaining the invalid input.

graphql.execution.**default_field_resolver**(*source*, *info*, ***args*)

Default field resolver.

If a resolve function is not given, then a default resolve behavior is used which takes the property of the source object of the same name as the field and returns it as the result, or if it's a function, returns the result of calling that function while passing along args and context.

For dictionaries, the field names are used as keys, for all other objects they are used as attribute names.

class graphql.execution.**ExecutionContext**(*schema: graphql.type.schema.GraphQLSchema, fragments: Dict[str, graphql.language.ast.FragmentDefinitionNode], root_value: Any, context_value: Any, operation: graphql.language.ast.OperationDefinitionNode, variable_values: Dict[str, Any], field_resolver: Callable[[...], Any], type_resolver: Callable[[Any, graphql.type.definition.GraphQLResolveInfo, GraphQLAbstractType], Union[Awaitable[Union[GraphQLObjectType, str, None]], GraphQLObjectType, str, None]], errors: List[graphql.error.graphql_error.GraphQLError], middleware_manager: Optional[graphql.execution.middleware.MiddlewareManager]*)

Data that must be available at all points during query execution.

Namely, schema of the type system that is currently executing, and the fragments defined in the query document.

class graphql.execution.**ExecutionResult**

The result of GraphQL execution.

- *data* is the result of a successful execution of the query.
- *errors* is included when any errors occurred as a non-empty list.

graphql.execution.**get_directive_values**(*directive_def: graphql.type.directives.GraphQLDirective, node: Union[graphql.language.ast.ExecutableDefinitionNode, graphql.language.ast.SelectionNode, graphql.language.ast.SchemaDefinitionNode, graphql.language.ast.TypeDefinitionNode, graphql.language.ast.TypeExtensionNode], variable_values: Dict[str, Any] = None*) → Optional[Dict[str, Any]]

Get coerced argument values based on provided nodes.

Prepares a dict of argument values given a directive definition and an AST node which may contain directives. Optionally also accepts a dict of variable values.

If the directive does not exist on the node, returns None.

Language

GraphQL Language

The *graphql.language* package is responsible for parsing and operating on the GraphQL language.

AST

class graphql.language.**Location**

AST Location

Contains a range of UTF-8 character offsets and token references that identify the region of the source from which the AST derived.

class graphql.language.**Node**(***kwargs*)

AST nodes

Each kind of AST node has its own class:

class graphql.language.**ArgumentNode**(***kwargs*)

class graphql.language.**BooleanValueNode**(***kwargs*)

class graphql.language.**DefinitionNode**(***kwargs*)

class graphql.language.**DirectiveDefinitionNode**(***kwargs*)

class graphql.language.**DirectiveNode**(***kwargs*)

class graphql.language.**DocumentNode**(***kwargs*)

class graphql.language.**EnumTypeDefinitionNode**(***kwargs*)

class graphql.language.**EnumTypeExtensionNode**(***kwargs*)

class graphql.language.**EnumValueDefinitionNode**(***kwargs*)

class graphql.language.**EnumValueNode**(***kwargs*)

class graphql.language.**ExecutableDefinitionNode**(***kwargs*)

class graphql.language.**FieldDefinitionNode**(***kwargs*)

class graphql.language.**FieldNode**(***kwargs*)

class graphql.language.**FloatValueNode**(***kwargs*)

class graphql.language.**FragmentDefinitionNode**(***kwargs*)

class graphql.language.**FragmentSpreadNode**(***kwargs*)

class graphql.language.**InlineFragmentNode**(***kwargs*)

class graphql.language.**InputObjectTypeDefinitionNode**(***kwargs*)

class graphql.language.**InputObjectTypeExtensionNode**(***kwargs*)

class graphql.language.**InputValueDefinitionNode**(***kwargs*)

class graphql.language.**IntValueNode**(***kwargs*)

class graphql.language.**InterfaceTypeDefinitionNode**(***kwargs*)

class graphql.language.**InterfaceTypeExtensionNode**(***kwargs*)

class graphql.language.**ListTypeNode**(***kwargs*)

class graphql.language.**ListValueNode**(***kwargs*)

class graphql.language.**NameNode**(***kwargs*)

class graphql.language.**NamedTypeNode**(***kwargs*)

class graphql.language.**NonNullTypeNode**(***kwargs*)

class graphql.language.**NullValueNode**(***kwargs*)

class graphql.language.**ObjectFieldNode**(***kwargs*)

class graphql.language.**ObjectTypeDefinitionNode**(***kwargs*)

class graphql.language.**ObjectTypeExtensionNode**(***kwargs*)

class graphql.language.**ObjectValueNode**(***kwargs*)

class graphql.language.**OperationDefinitionNode**(***kwargs*)

class graphql.language.**OperationType**
An enumeration.

class graphql.language.**OperationTypeDefinitionNode**(***kwargs*)

class graphql.language.**ScalarTypeDefinitionNode**(***kwargs*)

class graphql.language.**ScalarTypeExtensionNode**(***kwargs*)

class graphql.language.**SchemaDefinitionNode**(***kwargs*)

class graphql.language.**SchemaExtensionNode**(***kwargs*)

class graphql.language.**SelectionNode**(***kwargs*)

class graphql.language.**SelectionSetNode**(***kwargs*)

class graphql.language.**StringValueNode**(***kwargs*)

class graphql.language.**TypeDefinitionNode**(***kwargs*)

class graphql.language.**TypeExtensionNode**(***kwargs*)

class graphql.language.**TypeNode**(***kwargs*)

class graphql.language.**TypeSystemDefinitionNode**(***kwargs*)

graphql.language.**TypeSystemExtensionNode**

class graphql.language.**UnionTypeDefinitionNode**(***kwargs*)

class graphql.language.**UnionTypeExtensionNode**(***kwargs*)

class graphql.language.**ValueNode**(***kwargs*)

class graphql.language.**VariableDefinitionNode**(***kwargs*)

class graphql.language.**VariableNode**(***kwargs*)

Lexer

class graphql.language.**Lexer**(*source: graphql.language.source.Source*)
GraphQL Lexer

A Lexer is a stateful stream generator in that every time it is advanced, it returns the next token in the Source. Assuming the source lexes, the final Token emitted by the lexer will be of kind EOF, after which the lexer will repeatedly return the same EOF token whenever called.

class graphql.language.**TokenKind**
The different kinds of tokens that the lexer emits

class graphql.language.**Token**(*kind: graphql.language.token_kind.TokenKind*, *start: int*, *end: int*, *line: int*, *column: int*, *prev: Optional[graphql.language.ast.Token] = None*, *value: str = None*)

Location

graphql.language.**get_location**(*source: Source*, *position: int*) → graphql.language.location.SourceLocation
Get the line and column for a character position in the source.

Takes a Source and a UTF-8 character offset, and returns the corresponding line and column as a SourceLocation.

class graphql.language.**SourceLocation**
Represents a location in a Source.

graphql.language.**print_location**(*location: graphql.language.ast.Location*) → str
Render a helpful description of the location in the GraphQL Source document.

Parser

graphql.language.**parse**(*source: Union[graphql.language.source.Source, str], no_location=False, experimental_fragment_variables=False*) → graphql.language.ast.DocumentNode
Given a GraphQL source, parse it into a Document.

Throws GraphQLError if a syntax error is encountered.

By default, the parser creates AST nodes that know the location in the source that they correspond to. The *no_location* option disables that behavior for performance or testing.

Experimental features:

If *experimental_fragment_variables* is set to True, the parser will understand and parse variable definitions contained in a fragment definition. They'll be represented in the *variable_definitions* field of the *FragmentDefinitionNode*.

The syntax is identical to normal, query-defined variables. For example:

```
fragment A($var: Boolean = false) on T  {
  ...
}
```

graphql.language.**parse_type**(*source: Union[graphql.language.source.Source, str], no_location=False, experimental_fragment_variables=False*) → graphql.language.ast.TypeNode

Parse the AST for a given string containing a GraphQL Type.

Throws GraphQLError if a syntax error is encountered.

This is useful within tools that operate upon GraphQL Types directly and in isolation of complete GraphQL documents.

Consider providing the results to the utility function: *type_from_ast()*.

graphql.language.**parse_value**(*source: Union[graphql.language.source.Source, str], no_location=False, experimental_fragment_variables=False*) → graphql.language.ast.ValueNode

Parse the AST for a given string containing a GraphQL value.

Throws GraphQLError if a syntax error is encountered.

This is useful within tools that operate upon GraphQL Values directly and in isolation of complete GraphQL documents.

Consider providing the results to the utility function: *value_from_ast()*.

Source

class graphql.language.**Source**(*body: str, name: str = None, location_offset: graphql.language.location.SourceLocation = None*)

A representation of source input to GraphQL.

graphql.language.**print_source_location**(*source: graphql.language.source.Source, source_location: graphql.language.location.SourceLocation*) → str

Render a helpful description of the location in the GraphQL Source document.

Visitor

graphql.language.**visit**(*root: graphql.language.ast.Node, visitor: graphql.language.visitor.Visitor, visitor_keys=None*) → Any

Visit each node in an AST.

visit() will walk through an AST using a depth first traversal, calling the visitor's enter methods at each node in the traversal, and calling the leave methods after visiting that node and all of its child nodes.

By returning different values from the enter and leave methods, the behavior of the visitor can be altered, including skipping over a sub-tree of the AST (by returning False), editing the AST by returning a value or None to remove the value, or to stop the whole traversal by returning *BREAK*.

When using *visit()* to edit an AST, the original AST will not be modified, and a new version of the AST with the changes applied will be returned from the visit function.

To customize the node attributes to be used for traversal, you can provide a dictionary visitor_keys mapping node kinds to node attributes.

class graphql.language.**Visitor**
Visitor that walks through an AST.

Visitors can define two generic methods "enter" and "leave". The former will be called when a node is entered in the traversal, the latter is called after visiting the node and its child nodes. These methods have the following signature:

```
def enter(self, node, key, parent, path, ancestors):
    # The return value has the following meaning:
    # IDLE (None): no action
    # SKIP: skip visiting this node
    # BREAK: stop visiting altogether
    # REMOVE: delete this node
    # any other value: replace this node with the returned value
    return

def enter(self, node, key, parent, path, ancestors):
    # The return value has the following meaning:
    # IDLE (None) or SKIP: no action
    # BREAK: stop visiting altogether
    # REMOVE: delete this node
    # any other value: replace this node with the returned value
    return
```

The parameters have the following meaning:

Parameters

- **node** – The current node being visiting.
- **key** – The index or key to this node from the parent node or Array.
- **parent** – the parent immediately above this node, which may be an Array.
- **path** – The key path to get to this node from the root node.
- **ancestors** – All nodes and Arrays visited before reaching parent of this node. These correspond to array indices in *path*. Note: ancestors includes arrays which contain the parent of visited node.

You can also define node kind specific methods by suffixing them with an underscore followed by the kind of the node to be visited. For instance, to visit *field* nodes, you would defined the methods *enter_field()* and/or *leave_field()*, with the same signature as above. If no kind specific method has been defined for a given node, the generic method is called.

class graphql.language.**ParallelVisitor**(*visitors: Sequence[graphql.language.visitor.Visitor]*)
A Visitor which delegates to many visitors to run in parallel.

Each visitor will be visited for each node before moving on.

If a prior visitor edits a node, no following visitors will see that node.

class graphql.language.**TypeInfoVisitor**(*type_info: TypeInfo*, *visitor: graphql.language.visitor.Visitor*)
A visitor which maintains a provided TypeInfo.

The module also exports the following special symbols which can be used as return values in the *Visitor* methods to signal particular actions:

graphql.language.**BREAK = True**
This return value signals that no further nodes shall be visited.

graphql.language.**SKIP = False**
This return value signals that the current node shall be skipped.

graphql.language.**REMOVE = Ellipsis**
This return value signals that the current node shall be deleted.

graphql.language.**IDLE = None**
This return value signals that no additional action shall take place.

PyUtils

Python Utils

This package contains dependency-free Python utility functions used throughout the codebase.

Each utility should belong in its own file and be the default export.

These functions are not part of the module interface and are subject to change.

graphql.pyutils.**camel_to_snake**(*s*)
Convert from CamelCase to snake_case

graphql.pyutils.**snake_to_camel**(*s*, *upper=True*)
Convert from snake_case to CamelCase

If upper is set, then convert to upper CamelCase, otherwise the first character keeps its case.

graphql.pyutils.**cached_property**(*func*)
A cached property.

A property that is only computed once per instance and then replaces itself with an ordinary attribute. Deleting the attribute resets the property.

graphql.pyutils.**dedent**(*text: str*) → str
Fix indentation of given text by removing leading spaces and tabs.

Also removes leading newlines and trailing spaces and tabs, but keeps trailing newlines.

graphql.pyutils.**did_you_mean**(*suggestions: Sequence[str], sub_message: str = None*) → str
Given [A, B, C] return ' Did you mean A, B, or C?'

graphql.pyutils.**register_description**(*base: type*) → None
Register a class that shall be accepted as a description.

graphql.pyutils.**unregister_description**(*base: type*) → None
Unregister a class that shall no more be accepted as a description.

class graphql.pyutils.**EventEmitter**(*loop: Optional[asyncio.events.AbstractEventLoop] = None*)
A very simple EventEmitter.

add_listener(*event_name: str*, *listener: Callable*)
Add a listener.

emit(*event_name*, **args*, ***kwargs*)
Emit an event.

remove_listener(*event_name*, *listener*)
Removes a listener.

class graphql.pyutils.**EventEmitterAsyncIterator**(*event_emitter: graphql.pyutils.event_emitter.EventEmitter*, *event_name: str*)
Create an AsyncIterator from an EventEmitter.

Useful for mocking a PubSub system for tests.

graphql.pyutils.**identity_func**(*x: T = <INVALID>*, **_args*) → T
Return the first received argument.

graphql.pyutils.**inspect**(*value: Any*) → str
Inspect value and a return string representation for error messages.

Used to print values in error messages. We do not use repr() in order to not leak too much of the inner Python representation of unknown objects, and we do not use json.dumps() because not all objects can be serialized as JSON and we want to output strings with single quotes like Python repr() does it.

We also restrict the size of the representation by truncating strings and collections and allowing only a maximum recursion depth.

graphql.pyutils.**is_finite**(*value: Any*) → bool
Return true if a value is a finite number.

graphql.pyutils.**is_integer**(*value: Any*) → bool
Return true if a value is an integer number.

graphql.pyutils.**is_invalid**(*value: Any*) → bool
Return true if a value is undefined, or NaN.

graphql.pyutils.**is_nullish**(*value: Any*) → bool
Return true if a value is null, undefined, or NaN.

graphql.pyutils.**AwaitableOrValue**

graphql.pyutils.**suggestion_list**(*input_: str, options: Collection[str]*) → List[str]
Get list with suggestions for a given input.

Given an invalid input string and list of valid options, returns a filtered list of valid options sorted based on their similarity with the input.

class graphql.pyutils.**FrozenError**
Error when trying to change a frozen (read only) collection.

class graphql.pyutils.**FrozenList**
List that can only be read, but not changed.

class graphql.pyutils.**FrozenDict**
Dictionary that can only be read, but not changed.

class graphql.pyutils.**Path**
A generic path of string or integer indices

add_key(*key: Union[str, int]*) → graphql.pyutils.path.Path
Return a new Path containing the given key.

as_list() → List[Union[str, int]]
Return a list of the path keys.

key
current index in the path (string or integer)

prev
path with the previous indices

graphql.pyutils.**print_path_list**(*path: Sequence[Union[str, int]]*)
Build a string describing the path.

Subscription

GraphQL Subscription

The *graphql.subscription* package is responsible for subscribing to updates on specific data.

graphql.subscription.**subscribe**(*schema: graphql.type.schema.GraphQLSchema, document: graphql.language.ast.DocumentNode, root_value: Any = None, context_value: Any = None, variable_values: Dict[str, Any] = None, operation_name: str = None, field_resolver: Callable[[...], Any] = None, subscribe_field_resolver: Callable[[...], Any] = None*) → Union[AsyncIterator[graphql.execution.execute.ExecutionResult], graphql.execution.execute.ExecutionResult]

Create a GraphQL subscription.

Implements the "Subscribe" algorithm described in the GraphQL spec.

Returns a coroutine object which yields either an AsyncIterator (if successful) or an ExecutionResult (client error). The coroutine will raise an exception if a server error occurs.

If the client-provided arguments to this function do not result in a compliant subscription, a GraphQL Response (ExecutionResult) with descriptive errors and no data will be returned.

If the source stream could not be created due to faulty subscription resolver logic or underlying systems, the coroutine object will yield a single ExecutionResult containing *errors* and no *data*.

If the operation succeeded, the coroutine will yield an AsyncIterator, which yields a stream of ExecutionResults representing the response stream.

graphql.subscription.**create_source_event_stream**(*schema: graphql.type.schema.GraphQLSchema, document: graphql.language.ast.DocumentNode, root_value: Any = None, context_value: Any = None, variable_values: Dict[str, Any] = None, operation_name: str = None, field_resolver: Callable[[...], Any] = None*) → Union[AsyncIterable[Any], graphql.execution.execute.ExecutionResult]

Create source even stream

Implements the "CreateSourceEventStream" algorithm described in the GraphQL specification, resolving the subscription source event stream.

Returns a coroutine that yields an AsyncIterable.

If the client provided invalid arguments, the source stream could not be created, or the resolver did not return an AsyncIterable, this function will throw an error, which should be caught and handled by the caller.

A Source Event Stream represents a sequence of events, each of which triggers a GraphQL execution for that event.

This may be useful when hosting the stateful subscription service in a different process or machine than the stateless GraphQL execution engine, or otherwise separating these two steps. For more on this, see the "Supporting Subscriptions at Scale" information in the GraphQL spec.

Type

GraphQL Type System

The `graphql.type` package is responsible for defining GraphQL types and schema.

Definition

Predicates

graphql.type.**is_composite_type**(*type_: Any*) → bool

graphql.type.**is_enum_type**(*type_: Any*) → bool

graphql.type.**is_input_object_type**(*type_: Any*) → bool

graphql.type.**is_input_type**(*type_: Any*) → bool

graphql.type.**is_interface_type**(*type_: Any*) → bool

graphql.type.**is_leaf_type**(*type_: Any*) → bool

graphql.type.**is_list_type**(*type_: Any*) → bool

graphql.type.**is_named_type**(*type_: Any*) → bool

graphql.type.**is_non_null_type**(*type_: Any*) → bool

graphql.type.**is_nullable_type**(*type_: Any*) → bool

graphql.type.**is_object_type**(*type_: Any*) → bool

graphql.type.**is_output_type**(*type_: Any*) → bool

graphql.type.**is_scalar_type**(*type_: Any*) → bool

graphql.type.**is_type**(*type_: Any*) → bool

graphql.type.**is_union_type**(*type_: Any*) → bool

graphql.type.**is_wrapping_type**(*type_: Any*) → bool

Assertions

graphql.type.**assert_abstract_type**(*type_: Any*) → Union[graphql.type.definition.GraphQLInterfaceType, graphql.type.definition.GraphQLUnionType]

graphql.type.**assert_composite_type**(*type_: Any*) → graphql.type.definition.GraphQLType

graphql.type.**assert_enum_type**(*type_: Any*) → graphql.type.definition.GraphQLEnumType

graphql.type.**assert_input_object_type**(*type_: Any*) → graphql.type.definition.GraphQLInputObjectType

graphql.type.**assert_input_type**(*type_: Any*) → Union[graphql.type.definition.GraphQLScalarType, graphql.type.definition.GraphQLEnumType, graphql.type.definition.GraphQLInputObjectType, graphql.type.definition.GraphQLWrappingType]

graphql.type.**assert_interface_type**(*type_: Any*) → graphql.type.definition.GraphQLInterfaceType

graphql.type.**assert_leaf_type**(*type_: Any*) → Union[graphql.type.definition.GraphQLScalarType, graphql.type.definition.GraphQLEnumType]

graphql.type.**assert_list_type**(*type_: Any*) → graphql.type.definition.GraphQLList

graphql.type.**assert_named_type**(*type_: Any*) → graphql.type.definition.GraphQLNamedType

graphql.type.**assert_non_null_type**(*type_: Any*) → graphql.type.definition.GraphQLNonNull

graphql.type.**assert_nullable_type**(*type_: Any*) → Union[graphql.type.definition.GraphQLScalarType, graphql.type.definition.GraphQLObjectType, graphql.type.definition.GraphQLInterfaceType, graphql.type.definition.GraphQLUnionType, graphql.type.definition.GraphQLEnumType, graphql.type.definition.GraphQLInputObjectType, graphql.type.definition.GraphQLList]

graphql.type.**assert_object_type**(*type_: Any*) → graphql.type.definition.GraphQLObjectType

graphql.type.**assert_output_type**(*type_: Any*) → Union[graphql.type.definition.GraphQLScalarType, graphql.type.definition.GraphQLObjectType, graphql.type.definition.GraphQLInterfaceType, graphql.type.definition.GraphQLUnionType, graphql.type.definition.GraphQLEnumType, graphql.type.definition.GraphQLWrappingType]

graphql.type.**assert_scalar_type**(*type_: Any*) → graphql.type.definition.GraphQLScalarType

graphql.type.**assert_type**(*type_: Any*) → graphql.type.definition.GraphQLType

graphql.type.**assert_union_type**(*type_: Any*) → graphql.type.definition.GraphQLUnionType

graphql.type.**assert_wrapping_type**(*type_: Any*) → graphql.type.definition.GraphQLWrappingType

Un-modifiers

graphql.type.**get_nullable_type**(*type_*)
Unwrap possible non-null type

graphql.type.**get_named_type**(*type_*)
Unwrap possible wrapping type

Definitions

class graphql.type.**GraphQLEnumType**(*name: str, values: Union[Dict[str, GraphQLEnumValue], Dict[str, Any], Type[enum.Enum]], description: str = None, extensions: Dict[str, Any] = None, ast_node: graphql.language.ast.EnumTypeDefinitionNode = None, extension_ast_nodes: Sequence[graphql.language.ast.EnumTypeExtensionNode] = None*)

Enum Type Definition

Some leaf values of requests and input values are Enums. GraphQL serializes Enum values as strings, however internally Enums can be represented by any kind of type, often integers. They can also be provided as a Python Enum.

Example:

```
RGBType = GraphQLEnumType('RGB', {
    'RED': 0,
    'GREEN': 1,
```

(continues on next page)

(continued from previous page)

```
    'BLUE': 2
})
```

Example using a Python Enum:

```
class RGBEnum(enum.Enum):
    RED = 0
    GREEN = 1
    BLUE = 2

RGBType = GraphQLEnumType('RGB', enum.Enum)
```

Instead of raw values, you can also specify GraphQLEnumValue objects with more detail like description or deprecation information.

Note: If a value is not provided in a definition, the name of the enum value will be used as its internal value when the value is serialized.

class graphql.type.**GraphQLInputObjectType**(*name: str, fields: Union[Callable[[], Dict[str, GraphQLInputField]], Dict[str, GraphQLInputField]], description: str = None, out_type: Callable[[Dict[str, Any]], Any] = None, extensions: Dict[str, Any] = None, ast_node: graphql.language.ast.InputObjectTypeDefinitionNode = None, extension_ast_nodes: Sequence[graphql.language.ast.InputObjectTypeExtensionNode] = None*)

Input Object Type Definition

An input object defines a structured collection of fields which may be supplied to a field argument.

Using *NonNull* will ensure that a value must be provided by the query.

Example:

```
NonNullFloat = GraphQLNonNull(GraphQLFloat())

class GeoPoint(GraphQLInputObjectType):
    name = 'GeoPoint'
    fields = {
        'lat': GraphQLInputField(NonNullFloat),
        'lon': GraphQLInputField(NonNullFloat),
        'alt': GraphQLInputField(
                   GraphQLFloat(), default_value=0)
    }
```

The outbound values will be Python dictionaries by default, but you can have them converted to other types by specifying an *out_type* function or class.

class graphql.type.**GraphQLInterfaceType**(*name: str, fields: Union[Callable[[], Dict[str, graphql.type.definition.GraphQLField]], Dict[str, graphql.type.definition.GraphQLField]] = None, resolve_type: Callable[[Any, graphql.type.definition.GraphQLResolveInfo, GraphQLAbstractType], Union[Awaitable[Union[GraphQLObjectType, str, None]], GraphQLObjectType, str, None]] = None, description: str = None, extensions: Dict[str, Any] = None, ast_node: graphql.language.ast.InterfaceTypeDefinitionNode = None, extension_ast_nodes: Sequence[graphql.language.ast.InterfaceTypeExtensionNode] = None*)

Interface Type Definition

When a field can return one of a heterogeneous set of types, an Interface type is used to describe what types are possible, what fields are in common across all types, as well as a function to determine which type is actually used when the field is resolved.

Example:

```
EntityType = GraphQLInterfaceType('Entity', {
        'name': GraphQLField(GraphQLString),
    })
```

class graphql.type.**GraphQLObjectType**(*name: str, fields: Union[Callable[[], Dict[str, graphql.type.definition.GraphQLField]], Dict[str, graphql.type.definition.GraphQLField]], interfaces: Union[Callable[[], Sequence[GraphQLInterfaceType]], Sequence[GraphQLInterfaceType]] = None, is_type_of: Callable[[Any, graphql.type.definition.GraphQLResolveInfo], Union[Awaitable[bool], bool]] = None, extensions: Dict[str, Any] = None, description: str = None, ast_node: graphql.language.ast.ObjectTypeDefinitionNode = None, extension_ast_nodes: Sequence[graphql.language.ast.ObjectTypeExtensionNode] = None*)

Object Type Definition

Almost all of the GraphQL types you define will be object types. Object types have a name, but most importantly describe their fields.

Example:

```
AddressType = GraphQLObjectType('Address', {
    'street': GraphQLField(GraphQLString),
    'number': GraphQLField(GraphQLInt),
    'formatted': GraphQLField(GraphQLString,
        lambda obj, info, **args: f'{obj.number} {obj.street}')
})
```

When two types need to refer to each other, or a type needs to refer to itself in a field, you can use a lambda function with no arguments (a so-called "thunk") to supply the fields lazily.

Example:

```
PersonType = GraphQLObjectType('Person', lambda: {
    'name': GraphQLField(GraphQLString),
    'bestFriend': GraphQLField(PersonType)
})
```

class graphql.type.**GraphQLScalarType**(*name: str, serialize: Callable = None, parse_value: Callable = None, parse_literal: Callable = None, description: str = None, extensions: Dict[str, Any] = None, ast_node: graphql.language.ast.ScalarTypeDefinitionNode = None, extension_ast_nodes: Sequence[graphql.language.ast.ScalarTypeExtensionNode] = None*)

Scalar Type Definition

The leaf values of any request and input values to arguments are Scalars (or Enums) and are defined with a name and a series of functions used to parse input from ast or variables and to ensure validity.

If a type's serialize function does not return a value (i.e. it returns *None*), then no error will be included in the response.

Example:

def serialize_odd(value):

if value % 2 == 1: return value

odd_type = GraphQLScalarType('Odd', serialize=serialize_odd)

class graphql.type.**GraphQLUnionType**(*name, types: Union[Callable[[], Sequence[graphql.type.definition.GraphQLObjectType]], Sequence[graphql.type.definition.GraphQLObjectType]], resolve_type: Callable[[Any, graphql.type.definition.GraphQLResolveInfo, GraphQLAbstractType], Union[Awaitable[Union[GraphQLObjectType, str, None]], GraphQLObjectType, str, None]] = None, description: str = None, extensions: Dict[str, Any] = None, ast_node: graphql.language.ast.UnionTypeDefinitionNode = None, extension_ast_nodes: Sequence[graphql.language.ast.UnionTypeExtensionNode] = None*)

Union Type Definition

When a field can return one of a heterogeneous set of types, a Union type is used to describe what types are possible as well as providing a function to determine which type is actually used when the field is resolved.

Example:

class PetType(GraphQLUnionType): name = 'Pet' types = [DogType, CatType]

def resolve_type(self, value, _type):

if isinstance(value, Dog): return DogType()

if isinstance(value, Cat): return CatType()

Type Wrappers

class graphql.type.**GraphQLList**(*type_: GT*)

List Type Wrapper

A list is a wrapping type which points to another type. Lists are often created within the context of defining the fields of an object type.

Example:

```
class PersonType(GraphQLObjectType):
    name = 'Person'

    @property
    def fields(self):
        return {
            'parents': GraphQLField(GraphQLList(PersonType())),
            'children': GraphQLField(GraphQLList(PersonType())),
        }
```

class graphql.type.**GraphQLNonNull**(*type_: GNT*)

Non-Null Type Wrapper

A non-null is a wrapping type which points to another type. Non-null types enforce that their values are never null and can ensure an error is raised if this ever occurs during a request. It is useful for fields which you can make a strong guarantee on non-nullability, for example usually the id field of a database row will never be null.

Example:

```
class RowType(GraphQLObjectType):
    name = 'Row'
    fields = {
        'id': GraphQLField(GraphQLNonNull(GraphQLString()))
    }
```

Note: the enforcement of non-nullability occurs within the executor.

Types

graphql.type.**GraphQLAbstractType**

class graphql.type.**GraphQLArgument**(*type_: Union[graphql.type.definition.GraphQLScalarType, graphql.type.definition.GraphQLEnumType, graphql.type.definition.GraphQLInputObjectType, graphql.type.definition.GraphQLWrappingType], default_value: Any = <INVALID>, description: str = None, out_name: str = None, extensions: Dict[str, Any] = None, ast_node: graphql.language.ast.InputValueDefinitionNode = None*)

Definition of a GraphQL argument

graphql.type.**GraphQLArgumentMap**

graphql.type.**GraphQLCompositeType**

class graphql.type.**GraphQLEnumValue**(*value: Any = None*, *description: str = None*, *deprecation_reason: str = None*, *extensions: Dict[str, Any] = None*, *ast_node: graphql.language.ast.EnumValueDefinitionNode = None*)

graphql.type.**GraphQLEnumValueMap**

class graphql.type.**GraphQLField**(*type_: Union[graphql.type.definition.GraphQLScalarType, graphql.type.definition.GraphQLObjectType, graphql.type.definition.GraphQLInterfaceType, graphql.type.definition.GraphQLUnionType, graphql.type.definition.GraphQLEnumType, graphql.type.definition.GraphQLWrappingType], args: Dict[str, GraphQLArgument] = None, resolve: Optional[Callable[[...], Any]] = None, subscribe: Optional[Callable[[...], Any]] = None, description: str = None, deprecation_reason: str = None, extensions: Dict[str, Any] = None, ast_node: graphql.language.ast.FieldDefinitionNode = None*)

Definition of a GraphQL field

graphql.type.**GraphQLFieldMap**

class graphql.type.**GraphQLInputField**(*type_: Union[graphql.type.definition.GraphQLScalarType, graphql.type.definition.GraphQLEnumType, graphql.type.definition.GraphQLInputObjectType, graphql.type.definition.GraphQLWrappingType], default_value: Any = <INVALID>, description: str = None, out_name: str = None, extensions: Dict[str, Any] = None, ast_node: graphql.language.ast.InputValueDefinitionNode = None*)

Definition of a GraphQL input field

graphql.type.**GraphQLInputFieldMap**

graphql.type.**GraphQLInputType**

graphql.type.**GraphQLLeafType**

class graphql.type.**GraphQLNamedType**(*name: str*, *description: str = None*, *extensions: Dict[str, Any] = None*, *ast_node: graphql.language.ast.TypeDefinitionNode = None*, *extension_ast_nodes: Sequence[graphql.language.ast.TypeExtensionNode] = None*)

Base class for all GraphQL named types

graphql.type.**GraphQLNullableType**

graphql.type.**GraphQLOutputType**

class graphql.type.**GraphQLType**

Base class for all GraphQL types

class graphql.type.**GraphQLWrappingType**(*type_: GT*)

Base class for all GraphQL wrapping types

graphql.type.**Thunk**

Resolvers

graphql.type.**GraphQLFieldResolver**

graphql.type.**GraphQLIsTypeOfFn**

class graphql.type.**GraphQLResolveInfo**
Collection of information passed to the resolvers.

This is always passed as the first argument to the resolvers.

Note that contrary to the JavaScript implementation, the context (commonly used to represent an authenticated user, or request-specific caches) is included here and not passed as an additional argument.

graphql.type.**GraphQLTypeResolver**

Directives

Predicates

graphql.type.**is_directive**(*directive: Any*) → bool
Test if the given value is a GraphQL directive.

graphql.type.**is_specified_directive**(*directive: Any*) → bool
Check whether the given directive is one of the specified directives.

Definitions

class graphql.type.**GraphQLDirective**(*name: str, locations: Sequence[graphql.language.directive_locations.DirectiveLocation], args: Dict[str, graphql.type.definition.GraphQLArgument] = None, is_repeatable: bool = False, description: str = None, extensions: Dict[str, Any] = None, ast_node: graphql.language.ast.DirectiveDefinitionNode = None*)
GraphQL Directive

Directives are used by the GraphQL runtime as a way of modifying execution behavior. Type system creators will usually not create these directly.

graphql.type.**GraphQLIncludeDirective**

graphql.type.**GraphQLSkipDirective**

graphql.type.**GraphQLDeprecatedDirective**

graphql.type.**specified_directives = [<GraphQLDirective(@include)>, <GraphQLDirective(@skip)**
The full list of specified directives.

graphql.type.**DEFAULT_DEPRECATION_REASON = 'No longer supported'**
String constant that can be used as the default value for deprecation_reason.

Introspection

Predicates

graphql.type.**is_introspection_type**(*type_: Any*) → bool

Definitions

class graphql.type.**TypeKind**
An enumeration.

graphql.type.**TypeMetaFieldDef**

graphql.type.**TypeNameMetaFieldDef**

graphql.type.**SchemaMetaFieldDef**

graphql.type.**introspection_types = {'__Directive': <GraphQLObjectType '__Directive'>, '__**
A dictionary containing all introspection types.

Scalars

Predicates

graphql.type.**is_specified_scalar_type**(*type_: Any*) → bool

Definitions

graphql.type.**GraphQLBoolean**

graphql.type.**GraphQLFloat**

graphql.type.**GraphQLID**

graphql.type.**GraphQLInt**

graphql.type.**GraphQLString**

The list of all specified directives is available as *specified_directives*.

Schema

Predicates

graphql.type.**is_schema**(*schema: Any*) → bool
Test if the given value is a GraphQL schema.

Definitions

class graphql.type.**GraphQLSchema**(*query: graphql.type.definition.GraphQLObjectType = None*, *mutation: graphql.type.definition.GraphQLObjectType = None*, *subscription: graphql.type.definition.GraphQLObjectType = None*, *types: Sequence[graphql.type.definition.GraphQLNamedType] = None*, *directives: Sequence[graphql.type.directives.GraphQLDirective] = None*, *extensions: Dict[str, Any] = None*, *ast_node: graphql.language.ast.SchemaDefinitionNode = None*, *extension_ast_nodes: Sequence[graphql.language.ast.SchemaExtensionNode] = None*, *assume_valid: bool = False*)

Schema Definition

A Schema is created by supplying the root types of each type of operation, query and mutation (optional). A schema definition is then supplied to the validator and executor.

Example:

```
MyAppSchema = GraphQLSchema(
  query=MyAppQueryRootType,
  mutation=MyAppMutationRootType)
```

Note: When the schema is constructed, by default only the types that are reachable by traversing the root types are included, other types must be explicitly referenced.

Example:

```
character_interface = GraphQLInterfaceType('Character', ...)

human_type = GraphQLObjectType(
    'Human', interfaces=[character_interface], ...)

droid_type = GraphQLObjectType(
    'Droid', interfaces: [character_interface], ...)

schema = GraphQLSchema(
    query=GraphQLObjectType('Query',
        fields={'hero': GraphQLField(character_interface, ....)}),
    ...
    # Since this schema references only the `Character` interface it's
    # necessary to explicitly list the types that implement it if
    # you want them to be included in the final schema.
    types=[human_type, droid_type])
```

Note: If a list of *directives* are provided to GraphQLSchema, that will be the exact list of directives represented and allowed. If *directives* is not provided, then a default set of the specified directives (e.g. @include and @skip) will be used. If you wish to provide *additional* directives to these specified directives, you must explicitly declare them. Example:

```
MyAppSchema = GraphQLSchema(
  ...
  directives=specified_directives + [my_custom_directive])
```

Validate

Functions:

graphql.type.**validate_schema**(*schema: graphql.type.schema.GraphQLSchema*) → List[graphql.error.graphql_error.GraphQLError]

Validate a GraphQL schema.

Implements the "Type Validation" sub-sections of the specification's "Type System" section.

Validation runs synchronously, returning a list of encountered errors, or an empty list if no errors were encountered and the Schema is valid.

Assertions

graphql.type.**assert_valid_schema**(*schema: graphql.type.schema.GraphQLSchema*) → None

Utility function which asserts a schema is valid.

Throws a TypeError if the schema is invalid.

Utilities

GraphQL Utilities

The *graphql.utilities* package contains common useful computations to use with the GraphQL language and type objects.

The GraphQL query recommended for a full schema introspection:

graphql.utilities.**get_introspection_query**(*descriptions=True*) → str

Get a query for introspection, optionally without descriptions.

Get the target Operation from a Document:

graphql.utilities.**get_operation_ast**(*document_ast: graphql.language.ast.DocumentNode*, *operation_name: Optional[str] = None*) → Optional[graphql.language.ast.OperationDefinitionNode]

Get operation AST node.

Returns an operation AST given a document AST and optionally an operation name. If a name is not provided, an operation is only returned if only one is provided in the document.

Get the Type for the target Operation AST:

graphql.utilities.**get_operation_root_type**(*schema: graphql.type.schema.GraphQLSchema, operation: Union[graphql.language.ast.OperationDefinitionNode, graphql.language.ast.OperationTypeDefinitionNode]*) → graphql.type.definition.GraphQLObjectType

Extract the root type of the operation from the schema.

Convert a GraphQLSchema to an IntrospectionQuery:

graphql.utilities.**introspection_from_schema**(*schema: graphql.type.schema.GraphQLSchema*, *descriptions: bool = True*) → Dict[str, Any]

Build an IntrospectionQuery from a GraphQLSchema

IntrospectionQuery is useful for utilities that care about type and field relationships, but do not need to traverse through those relationships.

This is the inverse of build_client_schema. The primary use case is outside of the server context, for instance when doing schema comparisons.

Build a GraphQLSchema from an introspection result:

graphql.utilities.**build_client_schema**(*introspection: Dict[KT, VT], assume_valid: bool = False*) → graphql.type.schema.GraphQLSchema

Build a GraphQLSchema for use by client tools.

Given the result of a client running the introspection query, creates and returns a GraphQLSchema instance which can be then used with all GraphQL-core 3 tools, but cannot be used to execute a query, as introspection does not represent the "resolver", "parse" or "serialize" functions or any other server-internal mechanisms.

This function expects a complete introspection result. Don't forget to check the "errors" field of a server response before calling this function.

Build a GraphQLSchema from GraphQL Schema language:

graphql.utilities.**build_ast_schema**(*document_ast: graphql.language.ast.DocumentNode*, *assume_valid: bool = False*, *assume_valid_sdl: bool = False*) → graphql.type.schema.GraphQLSchema

Build a GraphQL Schema from a given AST.

This takes the ast of a schema document produced by the parse function in src/language/parser.py.

If no schema definition is provided, then it will look for types named Query and Mutation.

Given that AST it constructs a GraphQLSchema. The resulting schema has no resolve methods, so execution will use default resolvers.

When building a schema from a GraphQL service's introspection result, it might be safe to assume the schema is valid. Set *assume_valid* to True to assume the produced schema is valid. Set *assume_valid_sdl* to True to assume it is already a valid SDL document.

graphql.utilities.**build_schema**(*source: Union[str, graphql.language.source.Source], assume_valid=False, assume_valid_sdl=False, no_location=False, experimental_fragment_variables=False*) → graphql.type.schema.GraphQLSchema

Build a GraphQLSchema directly from a source document.

graphql.utilities.**get_description**(*node: graphql.language.ast.Node*) → Optional[str]

@deprecated: Given an ast node, returns its string description.

Extend an existing GraphQLSchema from a parsed GraphQL Schema language AST:

graphql.utilities.**extend_schema**(*schema: graphql.type.schema.GraphQLSchema*, *document_ast: graphql.language.ast.DocumentNode*, *assume_valid=False*, *assume_valid_sdl=False*) → graphql.type.schema.GraphQLSchema

Extend the schema with extensions from a given document.

Produces a new schema given an existing schema and a document which may contain GraphQL type extensions and definitions. The original schema will remain unaltered.

Because a schema represents a graph of references, a schema cannot be extended without effectively making an entire copy. We do not know until it's too late if subgraphs remain unchanged.

This algorithm copies the provided schema, applying extensions while producing the copy. The original schema remains unaltered.

When extending a schema with a known valid extension, it might be safe to assume the schema is valid. Set *assume_valid* to true to assume the produced schema is valid. Set *assume_valid_sdl* to True to assume it is already a valid SDL document.

Sort a GraphQLSchema: .. autofunction:: lexicographic_sort_schema

Print a GraphQLSchema to GraphQL Schema language:

graphql.utilities.**print_introspection_schema**(*schema: graphql.type.schema.GraphQLSchema*) → str

graphql.utilities.**print_schema**(*schema: graphql.type.schema.GraphQLSchema*) → str

graphql.utilities.**print_type**(*type_: graphql.type.definition.GraphQLNamedType*) → str

graphql.utilities.**print_value**(*value: Any, type_: Union[graphql.type.definition.GraphQLScalarType, graphql.type.definition.GraphQLEnumType, graphql.type.definition.GraphQLInputObjectType, graphql.type.definition.GraphQLWrappingType]*) → str

Convenience function for printing a Python value

Create a GraphQLType from a GraphQL language AST:

graphql.utilities.**type_from_ast**(*schema*, *type_node*)

Get the GraphQL type definition from an AST node.

Given a Schema and an AST node describing a type, return a GraphQLType definition which applies to that type. For example, if provided the parsed AST node for *[User]*, a GraphQLList instance will be returned, containing the type called "User" found in the schema. If a type called "User" is not found in the schema, then None will be returned.

Create a Python value from a GraphQL language AST with a type:

graphql.utilities.**value_from_ast**(*value_node: Optional[graphql.language.ast.ValueNode], type_: Union[graphql.type.definition.GraphQLScalarType, graphql.type.definition.GraphQLEnumType, graphql.type.definition.GraphQLInputObjectType, graphql.type.definition.GraphQLWrappingType], variables: Dict[str, Any] = None*) → Any

Produce a Python value given a GraphQL Value AST.

A GraphQL type must be provided, which will be used to interpret different GraphQL Value literals.

Returns *INVALID* when the value could not be validly coerced according to the provided type.

GraphQL Value | JSON Value | Python Value |
——————— | ————- | ———— |
Input Object | Object | dict |
List | Array | list |
Boolean | Boolean | bool |
String | String | str |
Int / Float | Number | int / float |
Enum Value | Mixed | Any |
NullValue | null | None |

Create a Python value from a GraphQL language AST without a type:

graphql.utilities.**value_from_ast_untyped**(*value_node: graphql.language.ast.ValueNode, variables: Dict[str, Any] = None*) → Any

Produce a Python value given a GraphQL Value AST.

Unlike *value_from_ast()*, no type is provided. The resulting Python value will reflect the provided GraphQL value AST.

GraphQL Value | JSON Value | Python Value |

——————— | ———— | ———— |
Input Object | Object | dict |
List | Array | list |
Boolean | Boolean | bool |
String / Enum | String | str |
Int / Float | Number | int / float |
Null | null | None |

Create a GraphQL language AST from a Python value:

graphql.utilities.**ast_from_value**(*value: Any, type_: Union[graphql.type.definition.GraphQLScalarType, graphql.type.definition.GraphQLEnumType, graphql.type.definition.GraphQLInputObjectType, graphql.type.definition.GraphQLWrappingType]*) → Optional[graphql.language.ast.ValueNode]

Produce a GraphQL Value AST given a Python value.

A GraphQL type must be provided, which will be used to interpret different Python values.

JSON Value | GraphQL Value |
———— | ——————— |
Object | Input Object |
Array | List |
Boolean | Boolean |
String | String / Enum Value |
Number | Int / Float |
Mixed | Enum Value |
null | NullValue |

A helper to use within recursive-descent visitors which need to be aware of the GraphQL type system:

class graphql.utilities.**TypeInfo**(*schema: graphql.type.schema.GraphQLSchema, get_field_def_fn: Callable[[graphql.type.schema.GraphQLSchema, graphql.type.definition.GraphQLType, graphql.language.ast.FieldNode], Optional[graphql.type.definition.GraphQLField]] = None, initial_type: graphql.type.definition.GraphQLType = None*)

Utility class for keeping track of type definitions.

TypeInfo is a utility class which, given a GraphQL schema, can keep track of the current field and type definitions at any point in a GraphQL document AST during a recursive descent by calling *enter(node)* and *leave(node)*.

Coerce a Python value to a GraphQL type, or produce errors:

graphql.utilities.**coerce_input_value**(*input_value: Any, type_: Union[graphql.type.definition.GraphQLScalarType, graphql.type.definition.GraphQLEnumType, graphql.type.definition.GraphQLInputObjectType, graphql.type.definition.GraphQLWrappingType], on_error: Callable[[List[Union[str, int]], Any, graphql.error.graphql_error.GraphQLError], None] = <function default_on_error>, path: graphql.pyutils.path.Path = None*) → Any

Coerce a Python value given a GraphQL Input Type.

Deprecated, use *coerce_input_value()*:

graphql.utilities.**coerce_value**(*input_value: Any, type_: Union[graphql.type.definition.GraphQLScalarType, graphql.type.definition.GraphQLEnumType, graphql.type.definition.GraphQLInputObjectType, graphql.type.definition.GraphQLWrappingType], blame_node: graphql.language.ast.Node = None, path: graphql.pyutils.path.Path = None*) → graphql.utilities.coerce_value.CoercedValue

Coerce a Python value given a GraphQL Type.

Deprecated. Use coerce_input_value() directly for richer information.

Concatenate multiple ASTs together:

graphql.utilities.**concat_ast**(*asts: Sequence[graphql.language.ast.DocumentNode]*) → graphql.language.ast.DocumentNode

Concat ASTs.

Provided a collection of ASTs, presumably each from different files, concatenate the ASTs together into batched AST, useful for validating many GraphQL source files which together represent one conceptual application.

Separate an AST into an AST per Operation:

graphql.utilities.**separate_operations**(*document_ast: graphql.language.ast.DocumentNode*) → Dict[str, graphql.language.ast.DocumentNode]

Separate operations in a given AST document.

This function accepts a single AST document which may contain many operations and fragments and returns a collection of AST documents each of which contains a single operation as well the fragment definitions it refers to.

Strip characters that are not significant to the validity or execution of a GraphQL document:

graphql.utilities.**strip_ignored_characters**(*source: Union[str, graphql.language.source.Source]*) → str

Strip characters that are ignored anyway.

Strips characters that are not significant to the validity or execution of a GraphQL document:

- UnicodeBOM
- WhiteSpace
- LineTerminator
- Comment
- Comma
- BlockString indentation

Note: It is required to have a delimiter character between neighboring non-punctuator tokes and this function always uses single space as delimiter.

It is guaranteed that both input and output documents if parsed would result in the exact same AST except for nodes location.

Warning: It is guaranteed that this function will always produce stable results. However, it's not guaranteed that it will stay the same between different releases due to bugfixes or changes in the GraphQL specification.

Query example:

```
query SomeQuery($foo: String!, $bar: String) {
  someField(foo: $foo, bar: $bar) {
    a
    b {
      c
      d
    }
  }
}
```

Becomes:

```
query SomeQuery($foo:String!$bar:String){someField(foo:$foo bar:$bar){a b{c d}}}
```

SDL example:

```
"""
Type description
"""
type Foo {
  """
  Field description
  """
  bar: String
}
```

Becomes:

```
"""Type description""" type Foo{"""Field description""" bar:String}
```

Comparators for types:

graphql.utilities.**is_equal_type**(*type_a: graphql.type.definition.GraphQLType*, *type_b: graphql.type.definition.GraphQLType*)

Check whether two types are equal.

Provided two types, return true if the types are equal (invariant).

graphql.utilities.**is_type_sub_type_of**(*schema: graphql.type.schema.GraphQLSchema*, *maybe_subtype: graphql.type.definition.GraphQLType*, *super_type: graphql.type.definition.GraphQLType*) → bool

Check whether a type is subtype of another type in a given schema.

Provided a type and a super type, return true if the first type is either equal or a subset of the second super type (covariant).

graphql.utilities.**do_types_overlap**(*schema*, *type_a*, *type_b*)

Check whether two types overlap in a given schema.

Provided two composite types, determine if they "overlap". Two composite types overlap when the Sets of possible concrete types for each intersect.

This is often used to determine if a fragment of a given type could possibly be visited in a context of another type.

This function is commutative.

Assert that a string is a valid GraphQL name:

graphql.utilities.**assert_valid_name**(*name: str*) → str
Uphold the spec rules about naming.

graphql.utilities.**is_valid_name_error**(*name: str*, *node: graphql.language.ast.Node = None*) → Optional[graphql.error.graphql_error.GraphQLError]
Return an Error if a name is invalid.

Compare two GraphQLSchemas and detect breaking changes:

graphql.utilities.**find_breaking_changes**(*old_schema: graphql.type.schema.GraphQLSchema*, *new_schema: graphql.type.schema.GraphQLSchema*) → List[graphql.utilities.find_breaking_changes.BreakingChange]
Find breaking changes.

Given two schemas, returns a list containing descriptions of all the types of breaking changes covered by the other functions down below.

graphql.utilities.**find_dangerous_changes**(*old_schema: graphql.type.schema.GraphQLSchema*, *new_schema: graphql.type.schema.GraphQLSchema*) → List[graphql.utilities.find_breaking_changes.DangerousChange]
Find dangerous changes.

Given two schemas, returns a list containing descriptions of all the types of potentially dangerous changes covered by the other functions down below.

class graphql.utilities.**BreakingChange**(*type*, *description*)

class graphql.utilities.**BreakingChangeType**
An enumeration.

class graphql.utilities.**DangerousChange**(*type*, *description*)

class graphql.utilities.**DangerousChangeType**
An enumeration.

Report all deprecated usages within a GraphQL document:

graphql.utilities.**find_deprecated_usages**(*schema: graphql.type.schema.GraphQLSchema*, *ast: graphql.language.ast.DocumentNode*) → List[graphql.error.graphql_error.GraphQLError]
Get a list of GraphQLError instances describing each deprecated use.

Validation

GraphQL Validation

The *graphql.validation* package fulfills the Validation phase of fulfilling a GraphQL result.

graphql.validation.**validate**(*schema: graphql.type.schema.GraphQLSchema*, *document_ast: graphql.language.ast.DocumentNode*, *rules: Sequence[Type[graphql.validation.rules.ASTValidationRule]] = None*, *type_info: graphql.utilities.type_info.TypeInfo = None*, *max_errors: int = None*) → List[graphql.error.graphql_error.GraphQLError]

Implements the "Validation" section of the spec.

Validation runs synchronously, returning a list of encountered errors, or an empty list if no errors were encountered and the document is valid.

A list of specific validation rules may be provided. If not provided, the default list of rules defined by the GraphQL specification will be used.

Each validation rule is a ValidationRule object which is a visitor object that holds a ValidationContext (see the language/visitor API). Visitor methods are expected to return GraphQLErrors, or lists of GraphQLErrors when invalid.

Optionally a custom TypeInfo instance may be provided. If not provided, one will be created from the provided schema.

class graphql.validation.**ASTValidationContext**(*ast: graphql.language.ast.DocumentNode*, *on_error: Callable[[graphql.error.graphql_error.GraphQLError], None] = None*)

Utility class providing a context for validation of an AST.

An instance of this class is passed as the context attribute to all Validators, allowing access to commonly useful contextual information from within a validation rule.

class graphql.validation.**ASTValidationRule**(*context: graphql.validation.validation_context.ASTValidationContext*)

Visitor for validation of an AST.

class graphql.validation.**SDLValidationContext**(*ast: graphql.language.ast.DocumentNode*, *schema: graphql.type.schema.GraphQLSchema = None*, *on_error: Callable[[graphql.error.graphql_error.GraphQLError], None] = None*)

Utility class providing a context for validation of an SDL AST.

An instance of this class is passed as the context attribute to all Validators, allowing access to commonly useful contextual information from within a validation rule.

class graphql.validation.**SDLValidationRule**(*context: graphql.validation.validation_context.SDLValidationContext*)

Visitor for validation of an SDL AST.

class graphql.validation.**ValidationContext**(*schema: graphql.type.schema.GraphQLSchema*, *ast: graphql.language.ast.DocumentNode*, *type_info: graphql.utilities.type_info.TypeInfo*, *on_error: Callable[[graphql.error.graphql_error.GraphQLError], None] = None*)

Utility class providing a context for validation using a GraphQL schema.

An instance of this class is passed as the context attribute to all Validators, allowing access to commonly useful contextual information from within a validation rule.

class graphql.validation.**ValidationRule**(*context: graphql.validation.validation_context.ValidationContext*)

Visitor for validation using a GraphQL schema.

Rules

This list includes all validation rules defined by the GraphQL spec. The order of the rules in this list has been adjusted to lead to the most clear output when encountering multiple validation errors:

graphql.validation.**specified_rules = FrozenList([...])**
This list includes all validation rules defined by the GraphQL spec.

The order of the rules in this list has been adjusted to lead to the most clear output when encountering multiple validation errors.

Spec Section: "Executable Definitions"

class graphql.validation.**ExecutableDefinitionsRule**(*context: graphql.validation.validation_context.ASTValidationConte*

Executable definitions

A GraphQL document is only valid for execution if all definitions are either operation or fragment definitions.

Spec Section: "Field Selections on Objects, Interfaces, and Unions Types"

class graphql.validation.**FieldsOnCorrectTypeRule**(*context: graphql.validation.validation_context.ValidationContext*)

Fields on correct type

A GraphQL document is only valid if all fields selected are defined by the parent type, or are an allowed meta field such as *__typename*.

Spec Section: "Fragments on Composite Types"

class graphql.validation.**FragmentsOnCompositeTypesRule**(*context: graphql.validation.validation_context.ValidationCon*

Fragments on composite type

Fragments use a type condition to determine if they apply, since fragments can only be spread into a composite type (object, interface, or union), the type condition must also be a composite type.

Spec Section: "Argument Names"

class graphql.validation.**KnownArgumentNamesRule**(*context: graphql.validation.validation_context.ValidationContext*)

Known argument names

A GraphQL field is only valid if all supplied arguments are defined by that field.

Spec Section: "Directives Are Defined"

class graphql.validation.**KnownDirectivesRule**(*context: Union[graphql.validation.validation_context.ValidationCo graphql.validation.validation_context.SDLValidationContext]*)

Known directives

A GraphQL document is only valid if all *@directives* are known by the schema and legally positioned.

Spec Section: "Fragment spread target defined"

class graphql.validation.**KnownFragmentNamesRule**(*context: graphql.validation.validation_context.ValidationContext*)

Known fragment names

A GraphQL document is only valid if all ... *Fragment* fragment spreads refer to fragments defined in the same document.

Spec Section: "Fragment Spread Type Existence"

class graphql.validation.**KnownTypeNamesRule**(*context: Union[graphql.validation.validation_context.ValidationCont graphql.validation.validation_context.SDLValidationContext]*)

Known type names

A GraphQL document is only valid if referenced types (specifically variable definitions and fragment conditions) are defined by the type schema.

Spec Section: "Lone Anonymous Operation"

class graphql.validation.**LoneAnonymousOperationRule**(*context: graphql.validation.validation_context.ASTValidationCont*

Lone anonymous operation

A GraphQL document is only valid if when it contains an anonymous operation (the query short-hand) that it contains only that one operation definition.

Spec Section: "Fragments must not form cycles"

class graphql.validation.**NoFragmentCyclesRule**(*context: graphql.validation.validation_context.ASTValidationCont*

No fragment cycles

Spec Section: "All Variable Used Defined"

class graphql.validation.**NoUndefinedVariablesRule**(*context: graphql.validation.validation_context.ValidationContext*)

No undefined variables

A GraphQL operation is only valid if all variables encountered, both directly and via fragment spreads, are defined by that operation.

Spec Section: "Fragments must be used"

class graphql.validation.**NoUnusedFragmentsRule**(*context: graphql.validation.validation_context.ASTValidationCo*

No unused fragments

A GraphQL document is only valid if all fragment definitions are spread within operations, or spread within other fragments spread within operations.

Spec Section: "All Variables Used"

class graphql.validation.**NoUnusedVariablesRule**(*context: graphql.validation.validation_context.ValidationContext*

No unused variables

A GraphQL operation is only valid if all variables defined by an operation are used, either directly or within a spread fragment.

Spec Section: "Field Selection Merging"

class graphql.validation.**OverlappingFieldsCanBeMergedRule**(*context: graphql.validation.validation_context.Validation*

Overlapping fields can be merged

A selection set is only valid if all fields (including spreading any fragments) either correspond to distinct response names or can be merged without ambiguity.

Spec Section: "Fragment spread is possible"

class graphql.validation.**PossibleFragmentSpreadsRule**(*context: graphql.validation.validation_context.ValidationContext*

Possible fragment spread

A fragment spread is only valid if the type condition could ever possibly be true: if there is a non-empty intersection of the possible parent types, and possible types which pass the type condition.

Spec Section: "Argument Optionality"

class graphql.validation.**ProvidedRequiredArgumentsRule**(*context: graphql.validation.validation_context.ValidationCon*

Provided required arguments

A field or directive is only valid if all required (non-null without a default value) field arguments have been provided.

Spec Section: "Leaf Field Selections"

class graphql.validation.**ScalarLeafsRule**(*context: graphql.validation.validation_context.ValidationContext*)

Scalar leafs

A GraphQL document is valid only if all leaf fields (fields without sub selections) are of scalar or enum types.

Spec Section: "Subscriptions with Single Root Field"

class graphql.validation.**SingleFieldSubscriptionsRule**(*context: graphql.validation.validation_context.ASTValidationC*

Subscriptions must only include one field.

A GraphQL subscription is valid only if it contains a single root.

Spec Section: "Argument Uniqueness"

class graphql.validation.**UniqueArgumentNamesRule**(*context: graphql.validation.validation_context.ASTValidationContext*)

Unique argument names

A GraphQL field or directive is only valid if all supplied arguments are uniquely named.

Spec Section: "Directives Are Unique Per Location"

class graphql.validation.**UniqueDirectivesPerLocationRule**(*context: Union[graphql.validation.validation_context.Vali graphql.validation.validation_context.SDLValidat*

Unique directive names per location

A GraphQL document is only valid if all non-repeatable directives at a given location are uniquely named.

Spec Section: "Fragment Name Uniqueness"

class graphql.validation.**UniqueFragmentNamesRule**(*context: graphql.validation.validation_context.ASTValidationContext*)

Unique fragment names

A GraphQL document is only valid if all defined fragments have unique names.

Spec Section: "Input Object Field Uniqueness"

class graphql.validation.**UniqueInputFieldNamesRule**(*context: graphql.validation.validation_context.ASTValidationConte*

Unique input field names

A GraphQL input object value is only valid if all supplied fields are uniquely named.

Spec Section: "Operation Name Uniqueness"

class graphql.validation.**UniqueOperationNamesRule**(*context: graphql.validation.validation_context.ASTValidationContext*

Unique operation names

A GraphQL document is only valid if all defined operations have unique names.

Spec Section: "Variable Uniqueness"

class graphql.validation.**UniqueVariableNamesRule**(*context: graphql.validation.validation_context.ASTValidationContext*)

Unique variable names

A GraphQL operation is only valid if all its variables are uniquely named.

Spec Section: "Value Type Correctness"

class graphql.validation.**ValuesOfCorrectTypeRule**(*context: graphql.validation.validation_context.ValidationContext*)

Value literals of correct type

A GraphQL document is only valid if all value literals are of the type expected at their position.

Spec Section: "Variables are Input Types"

class graphql.validation.**VariablesAreInputTypesRule**(*context: graphql.validation.validation_context.ValidationContext*)

Variables are input types

A GraphQL operation is only valid if all the variables it defines are of input types (scalar, enum, or input object).

Spec Section: "All Variable Usages Are Allowed"

class graphql.validation.**VariablesInAllowedPositionRule**(*context: graphql.validation.validation_context.ValidationCo*

Variables passed to field arguments conform to type

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index

E

F

G

I

K

L

M

N

O

P

R

S

T

U

V