Graphene

The documentation below is for the `dev` (prerelease) version of Graphene. To view the documentation for the latest stable Graphene version go to the v2 docs.

Contents:

- Schema
- Object types
- Fields (Mounted Types)
- Fields (Unmounted Types)
- GraphQL Scalars
- Graphene Scalars
- Enum
- Structures
- Type Extension
- Execution Metadata

Integrations

- Graphene-Django (source)
- Flask-Graphql (source)
- Graphene-SQLAlchemy (source)
- Graphene-GAE (source)
- Graphene-Mongo (source)
- Starlette (source)
- FastAPI (source)

NEXT: GETTING STARTED

Getting started

Introduction

What is GraphQL?

GraphQL is a query language for your API.

It provides a standard way to:

- *describe data provided by a server* in a statically typed **Schema**
- *request data* in a **Query** which exactly describes your data requirements and
- *receive data* in a **Response** containing only the data you requested.

For an introduction to GraphQL and an overview of its concepts, please refer to the official Graph

What is Graphene?

Graphene is a library that provides tools to implement a GraphQL API in Python using a *code-first*

Compare Graphene's *code-first* approach to building a GraphQL API with *schema-first* approache (Python). Instead of writing GraphQL **Schema Definition Language (SDL)**, we write Python code to

Graphene is fully featured with integrations for the most popular web frameworks and ORMs. Gra compliant with the GraphQL spec and provides tools and patterns for building a Relay-Compliant

An example in Graphene

Let's build a basic GraphQL schema to say "hello" and "goodbye" in Graphene.

When we send a **Query** requesting only one **Field**, `hello`, and specify a value for the `firstName`

```
{
  hello(firstName: "friend")
}
```

...we would expect the following Response containing only the data requested (the `goodbye` field

```
{
  "data": {
    "hello": "Hello friend!"
  }
}
```

Requirements

- Python (3.6, 3.7, 3.8, 3.9, 3.10, pypy)
- Graphene (3.0)

Project setup

```
pip install "graphene>=3.0"
```

Creating a basic Schema

In Graphene, we can define a simple schema using the following code:

```
from graphene import ObjectType, String, Schema

class Query(ObjectType):
    # this defines a Field `hello` in our Schema with a single Argument `fi
    # By default, the argument name will automatically be camel-based into
    hello = String(first_name=String(default_value="stranger"))
    goodbye = String()

    # our Resolver method takes the GraphQL context (root, info) as well as
    # Argument (first_name) for the Field and returns data for the query Re
    def resolve_hello(root, info, first_name):
        return f'Hello {first_name}!'

    def resolve_goodbye(root, info):
        return 'See ya!'

schema = Schema(query=Query)
```

A GraphQL **Schema** describes each **Field** in the data model provided by the server using scalar typ
types like *List* and *Object*. For more details refer to the Graphene Types Reference.

Our schema can also define any number of **Arguments** for our **Fields**. This is a powerful way for a
requirements for each **Field**.

For each **Field** in our **Schema**, we write a **Resolver** method to fetch data requested by a client's **Qu
Arguments. For more details, refer to this section on Resolvers.

Schema Definition Language (SDL)

In the GraphQL Schema Definition Language, we could describe the fields defined by our example

```
type Query {
  hello(firstName: String = "stranger"): String
  goodbye: String
}
```

Further examples in this documentation will use SDL to describe schema created by ObjectTypes

Querying

Then we can start querying our **Schema** by passing a GraphQL query string to `execute`:

```
# we can query for our field (with the default argument)
query_string = '{ hello }'
result = schema.execute(query_string)
print(result.data['hello'])
# "Hello stranger!"

# or passing the argument in the query
query_with_argument = '{ hello(firstName: "GraphQL") }'
result = schema.execute(query_with_argument)
print(result.data['hello'])
# "Hello GraphQL!"
```

Next steps

Congrats! You got your first Graphene schema working!

Normally, we don't need to directly execute a query string against our schema as Graphene provi
web frameworks like Flask and Django. Check out Integrations for more information on how to ge

PREVIOUS: GRAPHENE

Types Reference

PREVIOUS: GETTING STARTED
NEXT: SCHEMA

Execution

TABLE OF CONTENTS

Relay

Graphene has complete support for Relay and offers some utils to make integration from Python easy.

Useful links

- Getting started with Relay
- Relay Global Identification Specification
- Relay Cursor Connection Specification

Testing in Graphene

Automated testing is an extremely useful bug-killing tool for the modern developer use a collection of tests – a test suite – to solve, or avoid, a number of problems:

- When you're writing new code, you can use tests to validate your code wor expected.
- When you're refactoring or modifying old code, you can use tests to ensur changes haven't affected your application's behavior unexpectedly.

Testing a GraphQL application is a complex task, because a GraphQL application is several layers of logic – schema definition, schema validation, permissions and field

With Graphene test-execution framework and assorted utilities, you can simulate G requests, execute mutations, inspect your application's output and generally verify is doing what it should be doing.

Testing tools

Graphene provides a small set of tools that come in handy when writing tests.

Test Client

The test client is a Python class that acts as a dummy GraphQL client, allowing you views and interact with your Graphene-powered application programmatically.

Some of the things you can do with the test client are:

- Simulate Queries and Mutations and observe the response.
- Test that a given query request is rendered by a given Django template, wi template context that contains certain values.

Overview and a quick example

To use the test client, instantiate `graphene.test.Client` and retrieve GraphQL resp

```
from graphene.test import Client

def test_hey():
    client = Client(my_schema)
    executed = client.execute('''{ hey }''')
    assert executed == {
        'data': {
            'hey': 'hello!'
        }
    }
```

Execute parameters

You can also add extra keyword arguments to the `execute` method, such as `conte` `variables`, ...:

```
from graphene.test import Client

def test_hey():
    client = Client(my_schema)
    executed = client.execute('''{ hey }''', context={'user': '
    assert executed == {
        'data': {
            'hey': 'hello Peter!'
        }
    }
```

Snapshot testing

As our APIs evolve, we need to know when our changes introduce any breaking cha
might break some of the clients of our GraphQL app.

However, writing tests and replicating the same response we expect from our Grap
application can be a tedious and repetitive task, and sometimes it's easier to skip th

Because of that, we recommend the usage of SnapshotTest.

SnapshotTest lets us write all these tests in a breeze, as it automatically creates the
`snapshots` for us the first time the test are executed.

Here is a simple example on how our tests will look if we use `pytest`:

```
def test_hey(snapshot):
    client = Client(my_schema)
    # This will create a snapshot dir and a snapshot file
    # the first time the test is executed, with the response
    # of the execution.
    snapshot.assert_match(client.execute('''{ hey }'''))
```

If we are using `unittest`:

```
from snapshottest import TestCase

class APITestCase(TestCase):
    def test_api_me(self):
        """Testing the API for /me"""
        client = Client(my_schema)
        self.assertMatchSnapshot(client.execute('''{ hey }'''))
```

PREVIOUS: MUTATIONS

NEXT: API REF

API Reference

Schema

class `graphene.types.schema.Schema`(*query=None*, *mutation=None*, *subscription=None*, *types=Non*
auto_camelcase=True)[source]

Schema Definition. A Graphene Schema can execute operations (query, mutation, subscript
types. For advanced purposes, the schema can be used to lookup type definitions and answ
through introspection. :param query: Root query *ObjectType*. Describes entry point for field

data in your Schema.

Parameters:

- **mutation** (*Optional[Type[ObjectType]]*) – Root mutation *ObjectType*. De
to *create, update or delete* data in your API.
- **subscription** (*Optional[Type[ObjectType]]*) – Root subscription *ObjectT*
fields to receive continuous updates.
- **types** (*Optional[List[Type[ObjectType]]]*) – List of any types to include in
introspected through root types.
- **directives** (*List[GraphQLDirective], optional*) – List of custom directives
schema. Defaults to only include directives defined by GraphQL spec (@
[GraphQLIncludeDirective, GraphQLSkipDirective].
- **auto_camelcase** (*bool*) – Fieldnames will be transformed in Schema's Ty
camelCase (preferred by GraphQL standard). Default True.

`execute`(**args*, ***kwargs*)[source]

Execute a GraphQL query on the schema. Use the *graphql_sync* function from *graphql*
a query string. Most of the time this method will be called by one of the Graphene Integ
:param request_string: GraphQL request (query, mutation or subscription)

as string or parsed AST form from *graphql-core*.

Parameters:

- **root_value** (*Any, optional*) – Value to use as the parent value obje
- **context_value** (*Any, optional*) – Value to be made available to all r
be used to share authorization, dataloaders or other information
operation.
- **variable_values** (*dict, optional*) – If variables are used in the reque
provided in dictionary form mapping the variable name to the va
- **operation_name** (*str, optional*) – If multiple operations are provid
operation name must be provided for the result to be provided.
- **middleware** (*List[SupportsGraphQLMiddleware]*) – Supply reques
in *graphql-core*.
- **execution_context_class** (*ExecutionContext, optional*) – The execu
when resolving queries and mutations.

Returns: `ExecutionResult` containing any data and errors for the operation.

`execute_async`(**args*, ***kwargs*)[source]

Execute a GraphQL query on the schema asynchronously. Same as *execute*, but uses *g*
graphql_sync.

`subscribe`(*query*, **args*, ***kwargs*)[source]

Execute a GraphQL subscription on the schema asynchronously.

Object types

class `graphene.ObjectType`[source]

Object Type Definition

Almost all of the GraphQL types you define will be object types. Object types have a name, b
their fields.

The name of the type defined by an _ObjectType_ defaults to the class name. The type descr
docstring. This can be overridden by adding attributes to a Meta inner class.

The class attributes of an _ObjectType_ are mounted as instances of `graphene.Field`.

Methods starting with `resolve_<field_name>` are bound as resolvers of the matching Field n
provided, the default resolver is used.

Ambiguous types with Interface and Union can be determined through `is_type_of` method
attribute.

```
from graphene import ObjectType, String, Field

class Person(ObjectType):
    class Meta:
        description = 'A human'

    # implicitly mounted as Field
    first_name = String()
    # explicitly mounted as Field
    last_name = Field(String)

    def resolve_last_name(parent, info):
        return last_name
```

ObjectType must be mounted using `graphene.Field`.

```
from graphene import ObjectType, Field

class Query(ObjectType):

    person = Field(Person, description="My favorite person")
```

Meta class options (optional):

name (str): Name of the GraphQL type (must be unique in schema). Defaults to class
name.
description (str): Description of the GraphQL type in the schema. Defaults to class
docstring.
interfaces (Iterable[graphene.Interface]): GraphQL interfaces to extend with this object
all fields from interface will be included in this object's schema.
possible_types (Iterable[class]): Used to test parent value object via isinstance to see if
this type can be used to resolve an ambiguous type (interface, union).
default_resolver (any Callable resolver): Override the default resolver for this
type. Defaults to graphene default resolver which returns an attribute or dictiona
the field.
fields (Dict[str, graphene.Field]): Dictionary of field name to Field. Not recommended to
use (prefer class attributes).

An _ObjectType_ can be used as a simple value object by creating an instance of the class.

```
p = Person(first_name='Bob', last_name='Roberts')
assert p.first_name == 'Bob'
```

Parameters:

- ***args** (*List*[*Any*]) – Positional values to use for Field values of value obje
- **(Dict[str** (***kwargs*) – Any]): Keyword arguments to use for Field values

class `graphene.InputObjectType`(**args*, ***kwargs*)[source]

Input Object Type Definition

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

Using `graphene.NonNull` will ensure that a input value must be provided by the query.

All class attributes of `graphene.InputObjectType` are implicitly mounted as InputField using t

```
from graphene import InputObjectType, String, InputField

class Person(InputObjectType):
    # implicitly mounted as Input Field
    first_name = String(required=True)
    # explicitly mounted as Input Field
    last_name = InputField(String, description="Surname")
```

The fields on an input object type can themselves refer to input object types, but you can't n
your schema.

Meta class options (optional):

- name (str): the name of the GraphQL type (must be unique in schema). Defaults to clas
  name.
- description (str): the description of the GraphQL type in the schema. Defaults to class
  docstring.
- container (class): A class reference for a value object that allows for
  attribute initialization and access. Default InputObjectTypeContainer.
- fields (Dict[str, graphene.InputField]): Dictionary of field name to InputField. Not
  recommended to use (prefer class attributes).

class `graphene.Mutation` → None[source]

Object Type Definition (mutation field)

Mutation is a convenience type that helps us build a Field which takes Arguments and returr
ObjectType.

```
import graphene

class CreatePerson(graphene.Mutation):
    class Arguments:
        name = graphene.String()

    ok = graphene.Boolean()
    person = graphene.Field(Person)

    def mutate(parent, info, name):
        person = Person(name=name)
        ok = True
        return CreatePerson(person=person, ok=ok)

class Mutation(graphene.ObjectType):
    create_person = CreatePerson.Field()
```

Meta class options (optional):

- output (graphene.ObjectType): Or `Output` inner class with attributes on Mutation clas
  Or attributes from Mutation class. Fields which can be returned from this mutatio
- resolver (Callable resolver method): Or `mutate` method on Mutation class. Perform da
  change and return output.
- arguments (Dict[str, graphene.Argument]): Or `Arguments` inner class with attributes o
  Mutation class. Arguments to use for the mutation Field.
- name (str): Name of the GraphQL type (must be unique in schema). Defaults to class
  name.
- description (str): Description of the GraphQL type in the schema. Defaults to class
  docstring.
- interfaces (Iterable[graphene.Interface]): GraphQL interfaces to extend with the payloa
  object. All fields from interface will be included in this object's schema.
- fields (Dict[str, graphene.Field]): Dictionary of field name to Field. Not recommended to
  use (prefer class attributes or `Meta.output`).

classmethod `Field`(*name=None*, *description=None*, *deprecation_reason=None*, *required=Fa*

Mount instance of mutation Field.

Fields (Mounted Types)

class `graphene.Field`(*type_*, *args=None*, *resolver=None*, *source=None*, *deprecation_reason=None*,
description=None, *required=False*, *_creation_counter=None*, *default_value=None*, ***extra_args*)[s

Makes a field available on an ObjectType in the GraphQL schema. Any type can be mounted

- Object Type
- Scalar Type
- Enum
- Interface
- Union

All class attributes of `graphene.ObjectType` are implicitly mounted as Field using the below a

```
class Person(ObjectType):
    first_name = graphene.String(required=True)                # impli
    last_name = graphene.Field(String, description='Surname')  # expli
```

Parameters:

- **type** (*class for a graphene.UnmountedType*) – Must be a class (not an i graphene type (ex. scalar or object) which is used for the type of this fi
- **args** (*optional, Dict[str, graphene.Argument]*) – Arguments that can be use `**extra_args`, unless you use an argument name that clashes with presented here (see example).
- **resolver** (*optional, Callable*) – A function to get the value for a Field fron not set, the default resolver method for the schema is used.
- **source** (*optional, str*) – attribute name to resolve for this field from the Alternative to resolver (cannot set both source and resolver).
- **deprecation_reason** (*optional, str*) – Setting this value indicates that th may provide instruction or reason on how for clients to proceed.
- **required** (*optional, bool*) – indicates this field as not null in the graphql graphene.NonNull. Default False.
- **name** (*optional, str*) – the name of the GraphQL field (must be unique i attribute name.
- **description** (*optional, str*) – the description of the GraphQL field in the
- **default_value** (*optional, Any*) – Default value to resolve if none set from
- ****extra_args** (*optional, Dict[str, Union[graphene.Argument, graphene.* additional arguments to mount on the field.

class `graphene.Argument`(*type_*, *default_value=Undefined*, *deprecation_reason=None*, *description=* *required=False*, *_creation_counter=None*)[source]

Makes an Argument available on a Field in the GraphQL schema.

Arguments will be parsed and provided to resolver methods for fields as keyword argument

All `arg` and `**extra_args` for a `graphene.Field` are implicitly mounted as Argument using

```
from graphene import String, Boolean, Argument

age = String(
    # Boolean implicitly mounted as Argument
    dog_years=Boolean(description="convert to dog years"),
    # Boolean explicitly mounted as Argument
    decades=Argument(Boolean, default_value=False),
)
```

Parameters:

- **type** (*class for a graphene.UnmountedType*) – must be a class (not an i graphene type (ex. scalar or object) which is used for the type of this ar schema.
- **required** (*optional, bool*) – indicates this argument as not null in the gr behavior as graphene.NonNull. Default False.
- **name** (*optional, str*) – the name of the GraphQL argument. Defaults to
- **description** (*optional, str*) – the description of the GraphQL argument i
- **default_value** (*optional, Any*) – The value to be provided if the user doe operation.
- **deprecation_reason** (*optional, str*) – Setting this value indicates that th and may provide instruction or reason on how for clients to proceed. C is required (see spec).

class `graphene.InputField`(*type_*, *name=None*, *default_value=Undefined*, *deprecation_reason=Non* *required=False*, *_creation_counter=None*, ***extra_args*)[source]

Makes a field available on an ObjectType in the GraphQL schema. Any type can be mounted Interface and Union:

- Object Type
- Scalar Type
- Enum

Input object types also can't have arguments on their input fields, unlike regular `graphene.F`

All class attributes of `graphene.InputObjectType` are implicitly mounted as InputField using t

```
from graphene import InputObjectType, String, InputField

class Person(InputObjectType):
    # implicitly mounted as Input Field
    first_name = String(required=True)
    # explicitly mounted as Input Field
    last_name = InputField(String, description="Surname")
```

Parameters:

- **type** (*class for a graphene.UnmountedType*) – Must be a class (not an i graphene type (ex. scalar or object) which is used for the type of this fi
- **name** (*optional, str*) – Name of the GraphQL input field (must be uniqu attribute name.
- **default_value** (*optional, Any*) – Default value to use as input if none set mutation, etc.).
- **deprecation_reason** (*optional, str*) – Setting this value indicates that th may provide instruction or reason on how for clients to proceed.
- **description** (*optional, str*) – Description of the GraphQL field in the sch
- **required** (*optional, bool*) – Indicates this input field as not null in the gr validation error if argument not provided. Same behavior as graphene
- ****extra_args** (*optional, Dict*) – Not used.

Fields (Unmounted Types)

class `graphene.types.unmountedtype.UnmountedType`(**args*, ***kwargs*)[source]

This class acts a proxy for a Graphene Type, so it can be mounted dynamically as Field, Input

Instead of writing:

```
from graphene import ObjectType, Field, String

class MyObjectType(ObjectType):
    my_field = Field(String, description='Description here')
```

It lets you write:

```
from graphene import ObjectType, String

class MyObjectType(ObjectType):
    my_field = String(description='Description here')
```

It is not used directly, but is inherited by other types and streamlines their use in different c

- Object Type
- Scalar Type
- Enum
- Interface
- Union

An unmounted type will accept arguments based upon its context (ObjectType, Field or Inpu to the appropriate MountedType (Field, Argument or InputField).

See each Mounted type reference for more information about valid parameters.

GraphQL Scalars

class `graphene.Int`[source]

The *Int* scalar type represents non-fractional signed whole numeric values. Int can represen
and 2^53 - 1 since represented in JSON as double-precision floating point numbers specified
(http://en.wikipedia.org/wiki/IEEE_floating_point).

class `graphene.Float`[source]

The *Float* scalar type represents signed double-precision fractional values as specified by [IE
(http://en.wikipedia.org/wiki/IEEE_floating_point).

class `graphene.String`[source]

The *String* scalar type represents textual data, represented as UTF-8 character sequences. T
used by GraphQL to represent free-form human-readable text.

class `graphene.Boolean`[source]

The *Boolean* scalar type represents *true* or *false*.

class `graphene.ID`[source]

The *ID* scalar type represents a unique identifier, often used to refetch an object or as key fo
appears in a JSON response as a String; however, it is not intended to be human-readable. W
type, any string (such as *"4"*) or integer (such as *4*) input value will be accepted as an ID.

Graphene Scalars

class `graphene.Date`[source]

The *Date* scalar type represents a Date value as specified by [iso8601](https://en.wikipedia.o

class `graphene.DateTime`[source]

The *DateTime* scalar type represents a DateTime value as specified by [iso8601](https://en.w

class `graphene.Time`[source]

The *Time* scalar type represents a Time value as specified by [iso8601](https://en.wikipedia.o

class `graphene.Decimal`[source]

The *Decimal* scalar type represents a python Decimal.

class `graphene.UUID`[source]

Leverages the internal Python implementation of UUID (uuid.UUID) to provide native UUID o
input.

class `graphene.JSONString`[source]

Allows use of a JSON String for input / output from the GraphQL schema.

Use of this type is *not recommended* as you lose the benefits of having a defined, static sche
of GraphQL).

class `graphene.Base64`[source]

The *Base64* scalar type represents a base64-encoded String.

Enum

class `graphene.Enum`[source]

Enum type definition

Defines a static set of values that can be provided as a Field, Argument or InputField.

```
from graphene import Enum

class NameFormat(Enum):
    FIRST_LAST = "first_last"
    LAST_FIRST = "last_first"
```

Meta:

enum (optional, Enum): Python enum to use as a base for GraphQL Enum.

name (optional, str): Name of the GraphQL type (must be unique in schema). Defaults t

name.
description (optional, str): Description of the GraphQL type in the schema. Defaults to docstring.
deprecation_reason (optional, str): Setting this value indicates that the enum is depreciated and may provide instruction or reason on how for clients to proceed

Structures

class `graphene.List`(*of_type*, **args*, ***kwargs*)[source]

List Modifier

A list is a kind of type marker, a wrapping type which points to another type. Lists are often d defining the fields of an object type.

List indicates that many values will be returned (or input) for this field.

```
from graphene import List, String

field_name = List(String, description="There will be many values")
```

class `graphene.NonNull`(**args*, ***kwargs*)[source]

Non-Null Modifier

A non-null is a kind of type marker, a wrapping type which points to another type. Non-null t are never null and can ensure an error is raised if this ever occurs during a request. It is use make a strong guarantee on non-nullability, for example usually the id field of a database ro

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

NonNull can also be indicated on all Mounted types with the keyword argument `required`.

```
from graphene import NonNull, String

field_name = NonNull(String, description='This field will not be null'
another_field = String(required=True, description='This is equivalent
```

Type Extension

class `graphene.Interface`[source]

Interface Type Definition

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

```
from graphene import Interface, String

class HasAddress(Interface):
    class Meta:
        description = "Address fields"

    address1 = String()
    address2 = String()
```

If a field returns an Interface Type, the ambiguous type of the object can be determined usin and an ObjectType with `Meta.possible_types` or `is_type_of`.

Meta:

name (str): Name of the GraphQL type (must be unique in schema). Defaults to class name.
description (str): Description of the GraphQL type in the schema. Defaults to class docstring.
fields (Dict[str, graphene.Field]): Dictionary of field name to Field. Not recommended to use (prefer class attributes).

class `graphene.Union`[source]

Union Type Definition

When a field can return one of a heterogeneous set of types, a Union type is used to describ
well as providing a function to determine which type is actually used when the field is resolv

The schema in this example can take a search text and return any of the GraphQL object typ
or Starship.

Ambiguous return types can be resolved on each ObjectType through `Meta.possible_types`
method. Or by implementing `resolve_type` class method on the Union.

```
from graphene import Union, ObjectType, List

class SearchResult(Union):
    class Meta:
        types = (Human, Droid, Starship)

class Query(ObjectType):
    search = List(SearchResult.Field(
        search_text=String(description='Value to search for'))
    )
```

Meta:

types (Iterable[graphene.ObjectType]): Required. Collection of types that may be returr
by this Union for the graphQL schema.
name (optional, str): the name of the GraphQL type (must be unique in schema). Defau
name.
description (optional, str): the description of the GraphQL type in the schema. Defaults
docstring.

Execution Metadata

`graphene.ResolveInfo`

alias of `GraphQLResolveInfo`

class `graphene.Context`(***params*)[source]

Context can be used to make a convenient container for attributes to provide for execution
operation like a query.

```
from graphene import Context

context = Context(loaders=build_dataloaders(), request=my_web_request)
schema.execute('{ hello(name: "world") }', context=context)

def resolve_hello(parent, info, name):
    info.context.request  # value set in Context
    info.context.loaders  # value set in Context
    # ...
```

Parameters: ****params** (*Dict[str, Any]*) – values to make available on Context instance as attr

PREVIOUS: TESTING IN GRAPHENE

Schema

A GraphQL **Schema** defines the types and relationships between **Fields** in your API

A Schema is created by supplying the root ObjectType of each operation, query (m
mutation and subscription.

Schema will collect all type definitions related to the root operations and then sup
the validator and executor.

```
my_schema = Schema(
    query=MyRootQuery,
    mutation=MyRootMutation,
    subscription=MyRootSubscription
)
```

A Root Query is just a special ObjectType that defines the fields that are the entryp
your API. Root Mutation and Root Subscription are similar to Root Query, but for d
operation types:

- Query fetches data
- Mutation changes data and retrieves the changes
- Subscription sends changes to clients in real-time

Review the GraphQL documentation on Schema for a brief overview of fields, sche
operations.

Querying

To query a schema, call the `execute` method on it. See Executing a query for mor

```
query_string = 'query whoIsMyBestFriend { myBestFriend { lastNa
my_schema.execute(query_string)
```

Types

There are some cases where the schema cannot access all of the types that we pla
For example, when a field returns an `Interface`, the schema doesn't know about
implementations.

In this case, we need to use the `types` argument when creating the Schema:

```
my_schema = Schema(
    query=MyRootQuery,
    types=[SomeExtraObjectType, ]
)
```

Auto camelCase field names

By default all field and argument names (that are not explicitly set with the `name` a converted from `snake_case` to `camelCase` (as the API is usually being consumed b js/mobile client)

For example with the ObjectType the `last_name` field name is converted to `lastNa`

```
class Person(graphene.ObjectType):
    last_name = graphene.String()
    other_name = graphene.String(name='_other_Name')
```

In case you don't want to apply this transformation, provide a `name` argument to t constructor. `other_name` converts to `_other_Name` (without further transformatio

Your query should look like:

```
{
    lastName
    _other_Name
}
```

To disable this behavior, set the `auto_camelcase` to `False` upon schema instantia

```
my_schema = Schema(
    query=MyRootQuery,
    auto_camelcase=False,
)
```

PREVIOUS: TYPES REFERENCE

NEXT: SC

Scalars

Scalar types represent concrete values at the leaves of a query. There are several built in type
provides out of the box which represent common values in Python. You can also create your o
better express values that you might have in your data model.

All Scalar types accept the following arguments. All are optional:

`name`: *string*

Override the name of the Field.

`description`: *string*

A description of the type to show in the GraphiQL browser.

`required`: *boolean*

If `True`, the server will enforce a value for this field. See NonNull. Default is `False`.

`deprecation_reason`: *string*

Provide a deprecation reason for the Field.

`default_value`: *any*

Provide a default value for the Field.

Built in scalars

Graphene defines the following base Scalar Types that match the default GraphQL types:

`graphene.String`

Represents textual data, represented as UTF-8 character sequences. The String type is m
by GraphQL to represent free-form human-readable text.

`graphene.Int`

Represents non-fractional signed whole numeric values. Int is a signed 32-bit integer per
spec

`graphene.Float`

Represents signed double-precision fractional values as specified by IEEE 754.

`graphene.Boolean`

Represents *true* or *false*.

`graphene.ID`

Represents a unique identifier, often used to refetch an object or as key for a cache. The
in a JSON response as a String; however, it is not intended to be human-readable. When
input type, any string (such as *"4"*) or integer (such as *4*) input value will be accepted as a

---

Graphene also provides custom scalars for common values:

graphene.Date

Represents a Date value as specified by iso8601.

```
import datetime
from graphene import Schema, ObjectType, Date

class Query(ObjectType):
    one_week_from = Date(required=True, date_input=Date(required=True))

    def resolve_one_week_from(root, info, date_input):
        assert date_input == datetime.date(2006, 1, 2)
        return date_input + datetime.timedelta(weeks=1)

schema = Schema(query=Query)

results = schema.execute("""
    query {
        oneWeekFrom(dateInput: "2006-01-02")
    }
""")

assert results.data == {"oneWeekFrom": "2006-01-09"}
```

graphene.DateTime

Represents a DateTime value as specified by iso8601.

```
import datetime
from graphene import Schema, ObjectType, DateTime

class Query(ObjectType):
    one_hour_from = DateTime(required=True, datetime_input=DateTime(requ

    def resolve_one_hour_from(root, info, datetime_input):
        assert datetime_input == datetime.datetime(2006, 1, 2, 15, 4, 5)
        return datetime_input + datetime.timedelta(hours=1)

schema = Schema(query=Query)

results = schema.execute("""
    query {
        oneHourFrom(datetimeInput: "2006-01-02T15:04:05")
    }
""")

assert results.data == {"oneHourFrom": "2006-01-02T16:04:05"}
```

graphene.Time

Represents a Time value as specified by iso8601.

```
import datetime
from graphene import Schema, ObjectType, Time

class Query(ObjectType):
    one_hour_from = Time(required=True, time_input=Time(required=True))

    def resolve_one_hour_from(root, info, time_input):
        assert time_input == datetime.time(15, 4, 5)
        tmp_time_input = datetime.datetime.combine(datetime.date(1, 1, 1
        return (tmp_time_input + datetime.timedelta(hours=1)).time()

schema = Schema(query=Query)

results = schema.execute("""
    query {
        oneHourFrom(timeInput: "15:04:05")
    }
""")

assert results.data == {"oneHourFrom": "16:04:05"}
```

graphene.Decimal

Represents a Python Decimal value.

```
import decimal
from graphene import Schema, ObjectType, Decimal

class Query(ObjectType):
    add_one_to = Decimal(required=True, decimal_input=Decimal(required=T

    def resolve_add_one_to(root, info, decimal_input):
        assert decimal_input == decimal.Decimal("10.50")
        return decimal_input + decimal.Decimal("1")

schema = Schema(query=Query)

results = schema.execute("""
    query {
        addOneTo(decimalInput: "10.50")
    }
""")

assert results.data == {"addOneTo": "11.50"}
```

graphene.JSONString

Represents a JSON string.

```
from graphene import Schema, ObjectType, JSONString, String

class Query(ObjectType):
    update_json_key = JSONString(
        required=True,
        json_input=JSONString(required=True),
        key=String(required=True),
        value=String(required=True)
    )

    def resolve_update_json_key(root, info, json_input, key, value):
        assert json_input == {"name": "Jane"}
        json_input[key] = value
        return json_input

schema = Schema(query=Query)

results = schema.execute("""
    query {
        updateJsonKey(jsonInput: "{\\"name\\": \\"Jane\\"}", key: "name"
    }
""")

assert results.data == {"updateJsonKey": "{\"name\": \"Beth\"}"}
```

graphene.Base64

Represents a Base64 encoded string.

```
from graphene import Schema, ObjectType, Base64

class Query(ObjectType):
    increment_encoded_id = Base64(
        required=True,
        base64_input=Base64(required=True),
    )

    def resolve_increment_encoded_id(root, info, base64_input):
        assert base64_input == "4"
        return int(base64_input) + 1

schema = Schema(query=Query)

results = schema.execute("""
```

```
    query {
        incrementEncodedId(base64Input: "NA==")
    }
""")

assert results.data == {"incrementEncodedId": "NQ=="}
```

Custom scalars

You can create custom scalars for your schema. The following is an example for creating a Da

```
import datetime
from graphene.types import Scalar
from graphql.language import ast

class DateTime(Scalar):
    '''DateTime Scalar Description'''

    @staticmethod
    def serialize(dt):
        return dt.isoformat()

    @staticmethod
    def parse_literal(node, _variables=None):
        if isinstance(node, ast.StringValue):
            return datetime.datetime.strptime(
                node.value, "%Y-%m-%dT%H:%M:%S.%f")

    @staticmethod
    def parse_value(value):
        return datetime.datetime.strptime(value, "%Y-%m-%dT%H:%M:%S.%f")
```

Mounting Scalars

Scalars mounted in a `ObjectType`, `Interface` or `Mutation` act as `Field`s.

```
class Person(graphene.ObjectType):
    name = graphene.String()

# Is equivalent to:
class Person(graphene.ObjectType):
    name = graphene.Field(graphene.String)
```

Note: when using the `Field` constructor directly, pass the type and not an instance.

Types mounted in a `Field` act as `Argument`s.

```
graphene.Field(graphene.String, to=graphene.String())

# Is equivalent to:
graphene.Field(graphene.String, to=graphene.Argument(graphene.String))
```

PREVIOUS: SCHEMA

NEXT: LI

Lists and Non-Null

Object types, scalars, and enums are the only kinds of types you can define in Graphene. But when you use the types in other parts of the schema, or in your variable declarations, you can apply additional type modifiers that affect valida those values.

NonNull

```
import graphene

class Character(graphene.ObjectType):
    name = graphene.NonNull(graphene.String)
```

Here, we're using a `String` type and marking it as Non-Null by wrapping it usin `NonNull` class. This means that our server always expects to return a non-null for this field, and if it ends up getting a null value that will actually trigger a Gra execution error, letting the client know that something has gone wrong.

The previous `NonNull` code snippet is also equivalent to:

```
import graphene

class Character(graphene.ObjectType):
    name = graphene.String(required=True)
```

List

```
import graphene

class Character(graphene.ObjectType):
    appears_in = graphene.List(graphene.String)
```

Lists work in a similar way: We can use a type modifier to mark a type as a `List` indicates that this field will return a list of that type. It works the same for argu where the validation step will expect a list for that value.

NonNull Lists

By default items in a list will be considered nullable. To define a list without any items the type needs to be marked as `NonNull`. For example:

```
import graphene

class Character(graphene.ObjectType):
    appears_in = graphene.List(graphene.NonNull(graphene.Str
```

The above results in the type definition:

```
type Character {
    appearsIn: [String!]
}
```

PREVIOUS: SCALARS

NEXT: OBJECTT

ObjectType

A Graphene *ObjectType* is the building block used to define the relationship between **Fields** in you
retrieved.

The basics:

- Each ObjectType is a Python class that inherits from `graphene.ObjectType`.
- Each attribute of the ObjectType represents a `Field`.
- Each `Field` has a resolver method to fetch data (or Default Resolver).

Quick example

This example model defines a Person, with a first and a last name:

```
from graphene import ObjectType, String

class Person(ObjectType):
    first_name = String()
    last_name = String()
    full_name = String()

    def resolve_full_name(parent, info):
        return f"{parent.first_name} {parent.last_name}"
```

This *ObjectType* defines the field **first_name**, **last_name**, and **full_name**. Each field is specified as a
maps to a Field. Data is fetched by our `resolve_full_name` resolver method for `full_name` field a
fields.

The above `Person` ObjectType has the following schema representation:

```
type Person {
  firstName: String
  lastName: String
  fullName: String
}
```

Resolvers

A **Resolver** is a method that helps us answer **Queries** by fetching data for a **Field** in our **Schema**.

Resolvers are lazily executed, so if a field is not included in a query, its resolver will not be execute

Each field on an *ObjectType* in Graphene should have a corresponding resolver method to fetch
match the field name. For example, in the `Person` type above, the `full_name` field is resolved by

Each resolver method takes the parameters:

- Parent Value Object (parent) for the value object use to resolve most fields
- GraphQL Execution Info (info) for query and schema meta information and per-request
- GraphQL Arguments (**kwargs) as defined on the **Field**.

Resolver Parameters

Parent Value Object (*parent*)

This parameter is typically used to derive the values for most fields on an *ObjectType*.

The first parameter of a resolver method (*parent*) is the value object returned from the resolver o
parent field, such as a root Query field, then the value for *parent* is set to the `root_value` configu
(default `None`). See Executing a query for more details on executing queries.

Resolver example

If we have a schema with Person type and one field on the root query.

```
from graphene import ObjectType, String, Field

class Person(ObjectType):
    full_name = String()

    def resolve_full_name(parent, info):
        return f"{parent.first_name} {parent.last_name}"

class Query(ObjectType):
    me = Field(Person)

    def resolve_me(parent, info):
        # returns an object that represents a Person
        return get_human(name="Luke Skywalker")
```

When we execute a query against that schema.

```
schema = Schema(query=Query)

query_string = "{ me { fullName } }"
result = schema.execute(query_string)

assert result.data["me"] == {"fullName": "Luke Skywalker"}
```

Then we go through the following steps to resolve this query:

- `parent` is set with the root_value from query execution (None).
- `Query.resolve_me` called with `parent` None which returns a value object `Person("Luke",`
- This value object is then used as `parent` while calling `Person.resolve_full_name` to reso
  Skywalker".
- The scalar value is serialized and sent back in the query response.

Each resolver returns the next Parent Value Object (parent) to be used in executing the following
Scalar type, that value will be serialized and sent in the **Response**. Otherwise, while resolving Com
value be passed forward as the next Parent Value Object (parent).

Naming convention

This Parent Value Object (parent) is sometimes named `obj`, `parent`, or `source` in other Graph
named after the value object being resolved (ex. `root` for a root Query or Mutation, and `person`
Sometimes this argument will be named `self` in Graphene code, but this can be misleading due
executing queries in Graphene.

GraphQL Execution Info (*info*)

The second parameter provides two things:

- reference to meta information about the execution of the current GraphQL Query (fields
- access to per-request `context` which can be used to store user authentication, data loa
  useful for resolving the query.

Only context will be required for most applications. See Context for more information about setti

GraphQL Arguments (***kwargs*)

Any arguments that a field defines gets passed to the resolver function as keyword arguments. Fo

```
from graphene import ObjectType, Field, String

class Query(ObjectType):
    human_by_name = Field(Human, name=String(required=True))
```

```
    def resolve_human_by_name(parent, info, name):
        return get_human(name=name)
```

You can then execute the following query:

```
query {
    humanByName(name: "Luke Skywalker") {
        firstName
        lastName
    }
}
```

Note: There are several arguments to a field that are "reserved" by Graphene (see Fields (Mounte
argument that clashes with one of these fields by using the `args` parameter like so:

```
from graphene import ObjectType, Field, String

class Query(ObjectType):
    answer = String(args={'description': String()})

    def resolve_answer(parent, info, description):
        return description
```

Convenience Features of Graphene Resolvers

Implicit staticmethod

One surprising feature of Graphene is that all resolver methods are treated implicitly as staticme
other methods in Python, the first argument of a resolver is *never* `self` while it is being executed
argument is always Parent Value Object (parent). In practice, this is very convenient as, in GraphQ
concerned with the using the parent value object to resolve queries than attributes on the Pytho

The two resolvers in this example are effectively the same.

```
from graphene import ObjectType, String

class Person(ObjectType):
    first_name = String()
    last_name = String()

    @staticmethod
    def resolve_first_name(parent, info):
        '''
        Decorating a Python method with `staticmethod` ensures that `self`
        argument. However, Graphene does not need this decorator for this t
        '''
        return parent.first_name

    def resolve_last_name(parent, info):
        '''
        Normally the first argument for this method would be `self`, but Gr
        a staticmethod implicitly.
        '''
        return parent.last_name

    # ...
```

If you prefer your code to be more explicit, feel free to use `@staticmethod` decorators. Otherwise,
them!

Default Resolver

If a resolver method is not defined for a **Field** attribute on our *ObjectType*, Graphene supplies a d

If the Parent Value Object (parent) is a dictionary, the resolver will look for a dictionary key match
resolver will get the attribute from the parent value object matching the field name.

```
from collections import namedtuple
```

```
from graphene import ObjectType, String, Field, Schema

PersonValueObject = namedtuple("Person", ["first_name", "last_name"])

class Person(ObjectType):
    first_name = String()
    last_name = String()

class Query(ObjectType):
    me = Field(Person)
    my_best_friend = Field(Person)

    def resolve_me(parent, info):
        # always pass an object for `me` field
        return PersonValueObject(first_name="Luke", last_name="Skywalker")

    def resolve_my_best_friend(parent, info):
        # always pass a dictionary for `my_best_fiend_field`
        return {"first_name": "R2", "last_name": "D2"}

schema = Schema(query=Query)
result = schema.execute('''
    {
        me { firstName lastName }
        myBestFriend { firstName lastName }
    }
''')
# With default resolvers we can resolve attributes from an object..
assert result.data["me"] == {"firstName": "Luke", "lastName": "Skywalker"}

# With default resolvers, we can also resolve keys from a dictionary..
assert result.data["myBestFriend"] == {"firstName": "R2", "lastName": "D2"}
```

Advanced

GraphQL Argument defaults

If you define an argument for a field that is not required (and in a query execution it is not provide
passed to the resolver function at all. This is so that the developer can differentiate between a `un`
an explicit `null` value.

For example, given this schema:

```
from graphene import ObjectType, String

class Query(ObjectType):
    hello = String(required=True, name=String())

    def resolve_hello(parent, info, name):
        return name if name else 'World'
```

And this query:

```
query {
    hello
}
```

An error will be thrown:

```
TypeError: resolve_hello() missing 1 required positional argument: 'name'
```

You can fix this error in several ways. Either by combining all keyword arguments into a dict:

```
from graphene import ObjectType, String

class Query(ObjectType):
    hello = String(required=True, name=String())

    def resolve_hello(parent, info, **kwargs):
        name = kwargs.get('name', 'World')
        return f'Hello, {name}!'
```

Or by setting a default value for the keyword argument:

```
from graphene import ObjectType, String

class Query(ObjectType):
    hello = String(required=True, name=String())

    def resolve_hello(parent, info, name='World'):
        return f'Hello, {name}!'
```

One can also set a default value for an Argument in the GraphQL schema itself using Graphene!

```
from graphene import ObjectType, String

class Query(ObjectType):
    hello = String(
        required=True,
        name=String(default_value='World')
    )

    def resolve_hello(parent, info, name):
        return f'Hello, {name}!'
```

Resolvers outside the class

A field can use a custom resolver from outside the class:

```
from graphene import ObjectType, String

def resolve_full_name(person, info):
    return f"{person.first_name} {person.last_name}"

class Person(ObjectType):
    first_name = String()
    last_name = String()
    full_name = String(resolver=resolve_full_name)
```

Instances as value objects

Graphene `ObjectType`s can act as value objects too. So with the previous example you could use
of the *ObjectType*'s fields.

```
peter = Person(first_name='Peter', last_name='Griffin')

peter.first_name  # prints "Peter"
peter.last_name  # prints "Griffin"
```

Field camelcasing

Graphene automatically camelcases fields on *ObjectType* from `field_name` to `fieldName` to conf
Auto camelCase field names for more information.

ObjectType Configuration - Meta class

Graphene uses a Meta inner class on *ObjectType* to set different options.

GraphQL type name

By default the type name in the GraphQL schema will be the same as the class name that defines
changed by setting the `name` property on the `Meta` class:

```
from graphene import ObjectType

class MyGraphQlSong(ObjectType):
    class Meta:
        name = 'Song'
```

GraphQL Description

The schema description of an *ObjectType* can be set as a docstring on the Python object or on th

```
from graphene import ObjectType

class MyGraphQlSong(ObjectType):
    ''' We can set the schema description for an Object Type here on a docs
    class Meta:
        description = 'But if we set the description in Meta, this value is
```

Interfaces & Possible Types

Setting `interfaces` in Meta inner class specifies the GraphQL Interfaces that this Object impleme

Providing `possible_types` helps Graphene resolve ambiguous types such as interfaces or Unions

See Interfaces for more information.

```
from graphene import ObjectType, Node

Song = namedtuple('Song', ('title', 'artist'))

class MyGraphQlSong(ObjectType):
    class Meta:
        interfaces = (Node, )
        possible_types = (Song, )
```

PREVIOUS: LISTS AND NON-NULL

Enums

An `Enum` is a special `GraphQL` type that represents a set of symbolic names (members) bo
constant values.

Definition

You can create an `Enum` using classes:

```
import graphene

class Episode(graphene.Enum):
    NEWHOPE = 4
    EMPIRE = 5
    JEDI = 6
```

But also using instances of Enum:

```
Episode = graphene.Enum('Episode', [('NEWHOPE', 4), ('EMPIRE', 5), ('
```

Value descriptions

It's possible to add a description to an enum value, for that the enum value needs to have
property on it.

```
class Episode(graphene.Enum):
    NEWHOPE = 4
    EMPIRE = 5
    JEDI = 6

    @property
    def description(self):
        if self == Episode.NEWHOPE:
            return 'New Hope Episode'
        return 'Other episode'
```

Usage with Python Enums

In case the Enums are already defined it's possible to reuse them using the `Enum.from_enum`

```
graphene.Enum.from_enum(AlreadyExistingPyEnum)
```

`Enum.from_enum` supports a `description` and `deprecation_reason` lambdas as input so yo
description etc. to your enum without changing the original:

```
graphene.Enum.from_enum(
    AlreadyExistingPyEnum,
    description=lambda v: return 'foo' if v == AlreadyExistingPyEnum.
)
```

Notes

`graphene.Enum` uses `enum.Enum` internally (or a backport if that's not available) and can be
way, with the exception of member getters.

In the Python `Enum` implementation you can access a member by initing the Enum.

```
from enum import Enum

class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3

assert Color(1) == Color.RED
```

However, in Graphene `Enum` you need to call *.get* to have the same effect:

```
from graphene import Enum

class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3

assert Color.get(1) == Color.RED
```

PREVIOUS: OBJECTTYPE

NE

Interfaces

An *Interface* is an abstract type that defines a certain set of fields that a type must include to imp

For example, you can define an Interface `Character` that represents any character in the Star W

```
import graphene

class Character(graphene.Interface):
    id = graphene.ID(required=True)
    name = graphene.String(required=True)
    friends = graphene.List(lambda: Character)
```

Any ObjectType that implements `Character` will have these exact fields, with these arguments a

For example, here are some types that might implement `Character`:

```
class Human(graphene.ObjectType):
    class Meta:
        interfaces = (Character, )

    starships = graphene.List(Starship)
    home_planet = graphene.String()

class Droid(graphene.ObjectType):
    class Meta:
        interfaces = (Character, )

    primary_function = graphene.String()
```

Both of these types have all of the fields from the `Character` interface, but also bring in extra fie
`starships` and `primary_function`, that are specific to that particular type of character.

The full GraphQL schema definition will look like this:

```
interface Character {
    id: ID!
    name: String!
    friends: [Character]
}

type Human implements Character {
    id: ID!
    name: String!
    friends: [Character]
    starships: [Starship]
    homePlanet: String
}

type Droid implements Character {
    id: ID!
    name: String!
    friends: [Character]
    primaryFunction: String
}
```

Interfaces are useful when you want to return an object or set of objects, which might be of seve

For example, you can define a field `hero` that resolves to any `Character`, depending on the epis

```
class Query(graphene.ObjectType):
    hero = graphene.Field(
        Character,
```

```
        required=True,
        episode=graphene.Int(required=True)
    )

    def resolve_hero(root, info, episode):
        # Luke is the hero of Episode V
        if episode == 5:
            return get_human(name='Luke Skywalker')
        return get_droid(name='R2-D2')

schema = graphene.Schema(query=Query, types=[Human, Droid])
```

This allows you to directly query for fields that exist on the Character interface as well as selectin
type that implements the interface using inline fragments.

For example, the following query:

```
query HeroForEpisode($episode: Int!) {
    hero(episode: $episode) {
        __typename
        name
        ... on Droid {
            primaryFunction
        }
        ... on Human {
            homePlanet
        }
    }
}
```

Will return the following data with variables `{ "episode": 4 }`:

```
{
    "data": {
        "hero": {
            "__typename": "Droid",
            "name": "R2-D2",
            "primaryFunction": "Astromech"
        }
    }
}
```

And different data with the variables `{ "episode": 5 }`:

```
{
    "data": {
        "hero": {
            "__typename": "Human",
            "name": "Luke Skywalker",
            "homePlanet": "Tatooine"
        }
    }
}
```

Resolving data objects to types

As you build out your schema in Graphene it's common for your resolvers to return objects that
backing your GraphQL types rather than instances of the Graphene types (e.g. Django or SQLAlc
works well with `ObjectType` and `Scalar` fields, however when you start using Interfaces you mig
error:

```
"Abstract type Character must resolve to an Object type at runtime for fie
```

This happens because Graphene doesn't have enough information to convert the data object int
needed to resolve the `Interface`. To solve this you can define a `resolve_type` class method on
maps a data object to a Graphene type:

```python
class Character(graphene.Interface):
    id = graphene.ID(required=True)
    name = graphene.String(required=True)

    @classmethod
    def resolve_type(cls, instance, info):
        if instance.type == 'DROID':
            return Droid
        return Human
```

PREVIOUS: ENUMS

Unions

Union types are very similar to interfaces, but they don't get to specify any common fields between the types.

The basics:

- Each Union is a Python class that inherits from `graphene.Union`.
- Unions don't have any fields on it, just links to the possible ObjectTypes.

Quick example

This example model defines several ObjectTypes with their own fields. `SearchResult` is the implementation of `Union` of this object types.

```
import graphene

class Human(graphene.ObjectType):
    name = graphene.String()
    born_in = graphene.String()

class Droid(graphene.ObjectType):
    name = graphene.String()
    primary_function = graphene.String()

class Starship(graphene.ObjectType):
    name = graphene.String()
    length = graphene.Int()

class SearchResult(graphene.Union):
    class Meta:
        types = (Human, Droid, Starship)
```

Wherever we return a SearchResult type in our schema, we might get a Human, a Droid, or a Starship. Note that members of a union

type need to be concrete object types; you can't create a union type out of interfaces or other unions.

The above types have the following representation in a schema:

```
type Droid {
  name: String
  primaryFunction: String
}

type Human {
  name: String
  bornIn: String
}

type Ship {
  name: String
  length: Int
}

union SearchResult = Human | Droid | Starship
```

PREVIOUS: INTERFACES

NEXT: MUTATIONS

Mutations

A Mutation is a special ObjectType that also defines an Input.

Quick example

This example defines a Mutation:

```
import graphene

class CreatePerson(graphene.Mutation):
    class Arguments:
        name = graphene.String()

    ok = graphene.Boolean()
    person = graphene.Field(lambda: Person)

    def mutate(root, info, name):
        person = Person(name=name)
        ok = True
        return CreatePerson(person=person, ok=ok)
```

person and **ok** are the output fields of the Mutation when it is resolved.

Arguments attributes are the arguments that the Mutation `CreatePerson` needs for resolving, in this case **name** will be the only argument for the mutation.

mutate is the function that will be applied once the mutation is called. This method is just a special resolver that we can change data within. It takes the same arguments as the standard query Resolver Parameters.

So, we can finish our schema like this:

```
# ... the Mutation Class

class Person(graphene.ObjectType):
    name = graphene.String()
    age = graphene.Int()

class MyMutations(graphene.ObjectType):
    create_person = CreatePerson.Field()
```

```
# We must define a query for our schema
class Query(graphene.ObjectType):
    person = graphene.Field(Person)

schema = graphene.Schema(query=Query, mutation=MyMutations
```

Executing the Mutation

Then, if we query (`schema.execute(query_str)`) the following:

```
mutation myFirstMutation {
    createPerson(name:"Peter") {
        person {
            name
        }
        ok
    }
}
```

We should receive:

```
{
    "createPerson": {
        "person" : {
            "name": "Peter"
        },
        "ok": true
    }
}
```

InputFields and InputObjectTypes

InputFields are used in mutations to allow nested input data for mutations.

To use an InputField you define an InputObjectType that specifies the struct
of your input data:

```
import graphene

class PersonInput(graphene.InputObjectType):
    name = graphene.String(required=True)
    age = graphene.Int(required=True)

class CreatePerson(graphene.Mutation):
    class Arguments:
        person_data = PersonInput(required=True)

    person = graphene.Field(Person)

    def mutate(root, info, person_data=None):
        person = Person(
```

```
            name=person_data.name,
            age=person_data.age
        )
        return CreatePerson(person=person)
```

Note that **name** and **age** are part of **person_data** now.

Using the above mutation your new query would look like this:

```
mutation myFirstMutation {
    createPerson(personData: {name:"Peter", age: 24}) {
        person {
            name,
            age
        }
    }
}
```

InputObjectTypes can also be fields of InputObjectTypes allowing you to hav
as complex of input data as you need:

```
import graphene

class LatLngInput(graphene.InputObjectType):
    lat = graphene.Float()
    lng = graphene.Float()

#A location has a latlng associated to it
class LocationInput(graphene.InputObjectType):
    name = graphene.String()
    latlng = graphene.InputField(LatLngInput)
```

Output type example

To return an existing ObjectType instead of a mutation-specific type, set the
Output attribute to the desired ObjectType:

```
import graphene

class CreatePerson(graphene.Mutation):
    class Arguments:
        name = graphene.String()

    Output = Person

    def mutate(root, info, name):
        return Person(name=name)
```

Then, if we query (`schema.execute(query_str)`) with the following:

```
mutation myFirstMutation {
    createPerson(name:"Peter") {
        name
        __typename
    }
}
```

We should receive:

```
{
    "createPerson": {
        "name": "Peter",
        "__typename": "Person"
    }
}
```

PREVIOUS: UNIONS

NEXT: EXECUTION

Executing a query

For executing a query against a schema, you can directly call the `execute` me
on it.

```
from graphene import Schema

schema = Schema(...)
result = schema.execute('{ name }')
```

`result` represents the result of execution. `result.data` is the result of exec
the query, `result.errors` is `None` if no errors occurred, and is a non-empty
an error occurred.

Context

You can pass context to a query via `context`.

```
from graphene import ObjectType, String, Schema

class Query(ObjectType):
    name = String()

    def resolve_name(root, info):
        return info.context.get('name')

schema = Schema(Query)
result = schema.execute('{ name }', context={'name': 'Syrus
assert result.data['name'] == 'Syrus'
```

Variables

You can pass variables to a query via `variables`.

```
from graphene import ObjectType, Field, ID, Schema

class Query(ObjectType):
    user = Field(User, id=ID(required=True))

    def resolve_user(root, info, id):
```

```
        return get_user_by_id(id)

schema = Schema(Query)
result = schema.execute(
    '''
      query getUser($id: ID) {
        user(id: $id) {
          id
          firstName
          lastName
        }
      }
    ''',
    variables={'id': 12},
)
```

Root Value

Value used for Parent Value Object (parent) in root queries and mutations ca
overridden using `root` parameter.

```
from graphene import ObjectType, Field, Schema

class Query(ObjectType):
    me = Field(User)

    def resolve_user(root, info):
        return {'id': root.id, 'firstName': root.name}

schema = Schema(Query)
user_root = User(id=12, name='bob')
result = schema.execute(
    '''
    query getUser {
        user {
            id
            firstName
            lastName
        }
    }
    ''',
    root=user_root
)
assert result.data['user']['id'] == user_root.id
```

Operation Name

If there are multiple operations defined in a query string, `operation_name` sh
be used to indicate which should be executed.

```
from graphene import ObjectType, Field, Schema

class Query(ObjectType):
    user = Field(User)

    def resolve_user(root, info):
        return get_user_by_id(12)
```

```
schema = Schema(Query)
query_string = '''
    query getUserWithFirstName {
        user {
            id
            firstName
            lastName
        }
    }
    query getUserWithFullName {
        user {
            id
            fullName
        }
    }
'''
result = schema.execute(
    query_string,
    operation_name='getUserWithFullName'
)
assert result.data['user']['fullName']
```

PREVIOUS: EXECUTION

NEXT: MIDDLEWAR

Middleware

You can use `middleware` to affect the evaluation of fields in your schema.

A middleware is any object or function that responds to `resolve(next_middleware, *args)`

Inside that method, it should either:

- Send `resolve` to the next middleware to continue the evaluation; or
- Return a value to end the evaluation early.

Resolve arguments

Middlewares `resolve` is invoked with several arguments:

- `next` represents the execution chain. Call `next` to continue evaluation.
- `root` is the root value object passed throughout the query.
- `info` is the resolver info.
- `args` is the dict of arguments passed to the field.

Example

This middleware only continues evaluation if the `field_name` is not `'user'`

```
class AuthorizationMiddleware(object):
    def resolve(self, next, root, info, **args):
        if info.field_name == 'user':
            return None
        return next(root, info, **args)
```

And then execute it with:

```
result = schema.execute('THE QUERY', middleware=[AuthorizationMiddlew
```

If the `middleware` argument includes multiple middlewares, these middlewares will be ex
up, i.e. from last to first.

Functional example

Middleware can also be defined as a function. Here we define a middleware that logs the t
resolve each field:

```
from time import time as timer

def timing_middleware(next, root, info, **args):
    start = timer()
    return_value = next(root, info, **args)
    duration = round((timer() - start) * 1000, 2)
    parent_type_name = root._meta.name if root and hasattr(root, '_m
```

```
    logger.debug(f"{parent_type_name}.{info.field_name}: {duration} 
    return return_value
```

And then execute it with:

```
result = schema.execute('THE QUERY', middleware=[timing_middleware])
```

PREVIOUS: EXECUTING A QUERY

NEXT

Dataloader

DataLoader is a generic utility to be used as part of your application's data fetching lay
simplified and consistent API over various remote data sources such as databases or
via batching and caching. It is provided by a separate package *aiodataloader*
<https://pypi.org/project/aiodataloader/>.

Batching

Batching is not an advanced feature, it's DataLoader's primary feature. Create loaders
batch loading function.

```
from aiodataloader import DataLoader

class UserLoader(DataLoader):
    async def batch_load_fn(self, keys):
        # Here we call a function to return a user for each key in
        return [get_user(id=key) for key in keys]
```

A batch loading async function accepts a list of keys, and returns a list of `values`.

`DataLoader` will coalesce all individual loads which occur within a single frame of execu
(executed once the wrapping event loop is resolved) and then call your batch function
requested keys.

```
user_loader = UserLoader()

user1 = await user_loader.load(1)
user1_best_friend = await user_loader.load(user1.best_friend_id))

user2 = await user_loader.load(2)
user2_best_friend = await user_loader.load(user2.best_friend_id))
```

A naive application may have issued *four* round-trips to a backend for the required inf
with `DataLoader` this application will make at most *two*.

Note that loaded values are one-to-one with the keys and must have the same order.
that if you load all values from a single query, you must make sure that you then order
result for the results to match the keys:

```
class UserLoader(DataLoader):
    async def batch_load_fn(self, keys):
        users = {user.id: user for user in User.objects.filter(id_
        return [users.get(user_id) for user_id in keys]
```

`DataLoader` allows you to decouple unrelated parts of your application without sacrifi
performance of batch data-loading. While the loader presents an API that loads indivi

concurrent requests will be coalesced and presented to your batch loading function. T
your application to safely distribute data fetching requirements throughout your appl
maintain minimal outgoing data requests.

Using with Graphene

DataLoader pairs nicely well with Graphene/GraphQL. GraphQL fields are designed to
functions. Without a caching or batching mechanism, it's easy for a naive GraphQL ser
new database requests each time a field is resolved.

Consider the following GraphQL request:

```
{
  me {
    name
    bestFriend {
      name
    }
    friends(first: 5) {
      name
      bestFriend {
        name
      }
    }
  }
}
```

If `me`, `bestFriend` and `friends` each need to send a request to the backend, there c
most 13 database requests!

When using DataLoader, we could define the User type using our previous example wi
and at most 4 database requests, and possibly fewer if there are cache hits.

```
class User(graphene.ObjectType):
    name = graphene.String()
    best_friend = graphene.Field(lambda: User)
    friends = graphene.List(lambda: User)

    async def resolve_best_friend(root, info):
        return await user_loader.load(root.best_friend_id)

    async def resolve_friends(root, info):
        return await user_loader.load_many(root.friend_ids)
```

PREVIOUS: MIDDLEWARE

NEXT: FILE U

TABLE OF CONTENTS

File uploading

File uploading is not part of the official GraphQL spec yet and is not natively implemented in Graphene.

If your server needs to support file uploading then you can use the library: graphene-file-upload which enhances Graphene to add file uploads and conforms to the unoffical GraphQL multipart request spec.

PREVIOUS: DATALOADER
NEXT: SUBSCRIPTIONS

Subscriptions

To create a subscription, you can directly call the `subscribe` method on the schema. This method is async and must be awaited.

```
import asyncio
from datetime import datetime
from graphene import ObjectType, String, Schema, Field

# Every schema requires a query.
class Query(ObjectType):
    hello = String()

    def resolve_hello(root, info):
        return "Hello, world!"

class Subscription(ObjectType):
    time_of_day = String()

    async def subscribe_time_of_day(root, info):
        while True:
            yield datetime.now().isoformat()
            await asyncio.sleep(1)

schema = Schema(query=Query, subscription=Subscription)

async def main(schema):
    subscription = 'subscription { timeOfDay }'
    result = await schema.subscribe(subscription)
    async for item in result:
        print(item.data['timeOfDay'])

asyncio.run(main(schema))
```

The `result` is an async iterator which yields items in the same manner a a query.

Query Validation

GraphQL uses query validators to check if Query AST is valid and can be executed. Ever
server implements standard query validators. For example, there is an validator that t
field exists on queried type, that makes query fail with “Cannot query field on type” err

To help with common use cases, graphene provides a few validation rules out of the b

Depth limit Validator

The depth limit validator helps to prevent execution of malicious queries. It takes in th
arguments.

- `max_depth` is the maximum allowed depth for any operation in a GraphQL do
- `ignore` Stops recursive depth checking based on a field name. Either a string
  match the name, or a function that returns a boolean
- `callback` Called each time validation runs. Receives an Object which is a map
  for each operation.

Usage

Here is how you would implement depth-limiting on your schema.

```
from graphql import validate, parse
from graphene import ObjectType, Schema, String
from graphene.validation import depth_limit_validator


class MyQuery(ObjectType):
    name = String(required=True)


schema = Schema(query=MyQuery)

# queries which have a depth more than 20
# will not be executed.

validation_errors = validate(
    schema=schema.graphql_schema,
    document_ast=parse('THE QUERY'),
    rules=(
        depth_limit_validator(
            max_depth=20
        ),
    )
)
```

Disable Introspection

the disable introspection validation rule ensures that your schema cannot be introspe
useful security measure in production environments.

Usage

Here is how you would disable introspection for your schema.

```
from graphql import validate, parse
from graphene import ObjectType, Schema, String
from graphene.validation import DisableIntrospection


class MyQuery(ObjectType):
    name = String(required=True)


schema = Schema(query=MyQuery)

# introspection queries will not be executed.

validation_errors = validate(
    schema=schema.graphql_schema,
    document_ast=parse('THE QUERY'),
    rules=(
        DisableIntrospection,
    )
)
```

Implementing custom validators

All custom query validators should extend the ValidationRule base class importable fr
graphql.validation.rules module. Query validators are visitor classes. They are instanti
time of query validation with one required argument (context: ASTValidationContext).
perform validation, your validator class should define one or more of enter_* and leav
For possible enter/leave items as well as details on function documentation, please se
the visitor module. To make validation fail, you should call validator's report_error me
instance of GraphQLError describing failure reason. Here is an example query validato
field definitions in GraphQL query and fails query validation if any of those fields are bl

```
from graphql import GraphQLError
from graphql.language import FieldNode
from graphql.validation import ValidationRule


my_blacklist = (
    "disallowed_field",
)


def is_blacklisted_field(field_name: str):
    return field_name.lower() in my_blacklist


class BlackListRule(ValidationRule):
    def enter_field(self, node: FieldNode, *_args):
        field_name = node.name.value
        if not is_blacklisted_field(field_name):
            return

        self.report_error(
            GraphQLError(
                f"Cannot query '{field_name}': field is blacklist
            )
        )
```

Nodes

A `Node` is an Interface provided by `graphene.relay` that contains a single field `id` (which
object that inherits from it has to implement a `get_node` method for retrieving a `Node` b

Quick example

Example usage (taken from the Starwars Relay example):

```
class Ship(graphene.ObjectType):
    '''A ship in the Star Wars saga'''
    class Meta:
        interfaces = (relay.Node, )

    name = graphene.String(description='The name of the ship.')

    @classmethod
    def get_node(cls, info, id):
        return get_ship(id)
```

The `id` returned by the `Ship` type when you query it will be a scalar which contains eno
server to know its type and its id.

For example, the instance `Ship(id=1)` will return `U2hpcDox` as the id when you query it (v
base64 encoding of `Ship:1`), and which could be useful later if we want to query a node b

Custom Nodes

You can use the predefined `relay.Node` or you can subclass it, defining custom ways of h
encoded (using the `to_global_id` method in the class) or how we can retrieve a Node giv
(with the `get_node_from_global_id` method).

Example of a custom node:

```
class CustomNode(Node):

    class Meta:
        name = 'Node'

    @staticmethod
    def to_global_id(type_, id):
        return f"{type_}:{id}"

    @staticmethod
    def get_node_from_global_id(info, global_id, only_type=None):
        type_, id = global_id.split(':')
        if only_type:
            # We assure that the node type that we want to retrieve
            # is the same that was indicated in the field type
            assert type_ == only_type._meta.name, 'Received not compa

        if type_ == 'User':
            return get_user(id)
        elif type_ == 'Photo':
            return get_photo(id)
```

The `get_node_from_global_id` method will be called when `CustomNode.Field` is resolved.

Accessing node types

If we want to retrieve node instances from a `global_id` (scalar that identifies an instance and id), we can simply do `Node.get_node_from_global_id(info, global_id)`.

In the case we want to restrict the instance retrieval to a specific type, we can do: `Node.get_node_from_global_id(info, global_id, only_type=Ship)`. This will raise an error i doesn't correspond to a Ship type.

Node Root field

As is required in the Relay specification, the server must implement a root field called `no` `Node` Interface.

For this reason, `graphene` provides the field `relay.Node.Field`, which links to any type in which implements `Node`. Example usage:

```
class Query(graphene.ObjectType):
    # Should be CustomNode.Field() if we want to use our custom Node
    node = relay.Node.Field()
```

PREVIOUS: RELAY

NEXT

TABLE OF CONTENTS

Connection

A connection is a vitaminized version of a List that provides ways of slicing and paginating through it. The way you create Connection types in `graphene` is using `relay.Connection` and `relay.ConnectionField`.

Quick example

If we want to create a custom Connection on a given node, we have to subclass the `Connection` class.

In the following example, `extra` will be an extra field in the connection, and `other` an extra field in the Connection Edge.

```
class ShipConnection(Connection):
    extra = String()

    class Meta:
        node = Ship

    class Edge:
        other = String()
```

The `ShipConnection` connection class, will have automatically a `pageInfo` field, and a `edges` field (which is a list of `ShipConnection.Edge`). This `Edge` will have a `node` field linking to the specified node (in `ShipConnection.Meta`) and the field `other` that we defined in the class.

Connection Field

You can create connection fields in any Connection, in case any ObjectType that implements `Node` will have a default Connection.

```python
class Faction(graphene.ObjectType):
    name = graphene.String()
    ships = relay.ConnectionField(ShipConnection)

    def resolve_ships(root, info):
        return []
```

PREVIOUS: NODES

NEXT: MUTATIONS

Mutations

Most APIs don't just allow you to read data, they also allow you to write.

In GraphQL, this is done using mutations. Just like queries, Relay puts some additior requirements on mutations, but Graphene nicely manages that for you. All you nee make your mutation a subclass of `relay.ClientIDMutation`.

```
class IntroduceShip(relay.ClientIDMutation):

    class Input:
        ship_name = graphene.String(required=True)
        faction_id = graphene.String(required=True)

    ship = graphene.Field(Ship)
    faction = graphene.Field(Faction)

    @classmethod
    def mutate_and_get_payload(cls, root, info, **input):
        ship_name = input.ship_name
        faction_id = input.faction_id
        ship = create_ship(ship_name, faction_id)
        faction = get_faction(faction_id)
        return IntroduceShip(ship=ship, faction=faction)
```

Accepting Files

Mutations can also accept files, that's how it will work with different integrations:

```
class UploadFile(graphene.ClientIDMutation):
     class Input:
         pass
         # nothing needed for uploading file

     # your return fields
     success = graphene.String()

    @classmethod
    def mutate_and_get_payload(cls, root, info, **input):
        # When using it in Django, context will be the request
        files = info.context.FILES
        # Or, if used in Flask, context will be the flask global
        # files = context.files

        # do something with files

        return UploadFile(success=True)
```

PREVIOUS: CONNECTION

NEXT: TESTING IN GR