Graphene Documentation

Release 1.0.dev

Syrus Akbary

Dec 31, 2020

Contents

Welcome to the Graphene-Django docs.

Graphene-Django is built on top of Graphene. Graphene-Django provides some additional abstractions that make it easy to add GraphQL functionality to your Django project.

First time? We recommend you start with the installation guide to get set up and the basic tutorial. It is worth reading the core graphene docs to familiarize yourself with the basic utilities.

CHAPTER 1

Core tenets

If you want to expose your data through GraphQL - read the `Installation`, `Schema` and `Queries` section.

For more advanced use, check out the Relay tutorial.

1.1 Installation

Graphene-Django takes a few seconds to install and set up.

1.1.1 Requirements

Graphene-Django currently supports the following versions of Django:

- >= Django 1.11

1.1.2 Installation

```
pip install graphene-django
```

We strongly recommend pinning against a specific version of Graphene-Django because new versions could introduce breaking changes to your project.

Add `graphene_django` to the `INSTALLED_APPS` in the `settings.py` file of your Django project:

```
INSTALLED_APPS = [
    ...
    "django.contrib.staticfiles", # Required for GraphiQL
    "graphene_django"
]
```

We need to add a `graphql` URL to the `urls.py` of your Django project:

For Django 1.11:

```
from django.conf.urls import url
from graphene_django.views import GraphQLView

urlpatterns = [
    # ...
    url(r"graphql", GraphQLView.as_view(graphiql=True)),
]
```

For Django 2.0 and above:

```
from django.urls import path
from graphene_django.views import GraphQLView

urlpatterns = [
    # ...
    path("graphql", GraphQLView.as_view(graphiql=True)),
]
```

(Change `graphiql=True` to `graphiql=False` if you do not want to use the GraphiQL API browser.)

Finally, define the schema location for Graphene in the `settings.py` file of your Django project:

```
GRAPHENE = {
    "SCHEMA": "django_root.schema.schema"
}
```

Where `path.schema.schema` is the location of the `Schema` object in your Django project.

The most basic `schema.py` looks like this:

```
import graphene

class Query(graphene.ObjectType):
    hello = graphene.String(default_value="Hi!")

schema = graphene.Schema(query=Query)
```

To learn how to extend the schema object for your project, read the basic tutorial.

1.1.3 CSRF exempt

If you have enabled CSRF protection in your Django app you will find that it prevents your API clients from POSTing to the `graphql` endpoint. You can either update your API client to pass the CSRF token with each request (the Django docs have a guide on how to do that: https://docs.djangoproject.com/en/3.0/ref/csrf/#ajax) or you can exempt your Graphql endpoint from CSRF protection by wrapping the `GraphQLView` with the `csrf_exempt` decorator:

```
# urls.py

from django.urls import path
from django.views.decorators.csrf import csrf_exempt

from graphene_django.views import GraphQLView

urlpatterns = [
```

```
    # ...
    path("graphql", csrf_exempt(GraphQLView.as_view(graphiql=True))),
]
```

1.2 Basic Tutorial

Graphene Django has a number of additional features that are designed to make working with Django easy. Our primary focus in this tutorial is to give a good understanding of how to connect models from Django ORM to Graphene object types.

1.2.1 Set up the Django project

We will set up the project, create the following:

- A Django project called `cookbook`
- An app within `cookbook` called `ingredients`

```
# Create the project directory
mkdir cookbook
cd cookbook

# Create a virtualenv to isolate our package dependencies locally
virtualenv env
source env/bin/activate  # On Windows use `env\Scripts\activate`

# Install Django and Graphene with Django support
pip install django graphene_django

# Set up a new project with a single application
django-admin startproject cookbook .  # Note the trailing '.' character
cd cookbook
django-admin startapp ingredients
```

Now sync your database for the first time:

```
python manage.py migrate
```

Let's create a few simple models...

Defining our models

Let's get started with these models:

```
# cookbook/ingredients/models.py
from django.db import models

class Category(models.Model):
    name = models.CharField(max_length=100)

    def __str__(self):
        return self.name
```

```
class Ingredient(models.Model):
    name = models.CharField(max_length=100)
    notes = models.TextField()
    category = models.ForeignKey(
        Category, related_name="ingredients", on_delete=models.CASCADE
    )

    def __str__(self):
        return self.name
```

Add ingredients as INSTALLED_APPS:

```
# cookbook/settings.py

INSTALLED_APPS = [
    ...
    # Install the ingredients app
    "cookbook.ingredients",
]
```

Don't forget to create & run migrations:

```
python manage.py makemigrations
python manage.py migrate
```

Load some test data

Now is a good time to load up some test data. The easiest option will be to download the ingredients.json fixture and place it in `cookbook/ingredients/fixtures/ingredients.json`. You can then run the following:

```
python manage.py loaddata ingredients

Installed 6 object(s) from 1 fixture(s)
```

Alternatively you can use the Django admin interface to create some data yourself. You'll need to run the development server (see below), and create a login for yourself too (`python manage.py createsuperuser`).

Register models with admin panel:

```
# cookbook/ingredients/admin.py
from django.contrib import admin
from cookbook.ingredients.models import Category, Ingredient

admin.site.register(Category)
admin.site.register(Ingredient)
```

1.2.2 Hello GraphQL - Schema and Object Types

In order to make queries to our Django project, we are going to need few things:

- Schema with defined object types
- A view, taking queries as input and returning the result

GraphQL presents your objects to the world as a graph structure rather than a more hierarchical structure to which you may be accustomed. In order to create this representation, Graphene needs to know about each *type* of object which will appear in the graph.

This graph also has a *root type* through which all access begins. This is the `Query` class below.

To create GraphQL types for each of our Django models, we are going to subclass the `DjangoObjectType` class which will automatically define GraphQL fields that correspond to the fields on the Django models.

After we've done that, we will list those types as fields in the `Query` class.

Create `cookbook/schema.py` and type the following:

```
# cookbook/schema.py
import graphene
from graphene_django import DjangoObjectType

from cookbook.ingredients.models import Category, Ingredient

class CategoryType(DjangoObjectType):
    class Meta:
        model = Category
        fields = ("id", "name", "ingredients")

class IngredientType(DjangoObjectType):
    class Meta:
        model = Ingredient
        fields = ("id", "name", "notes", "category")

class Query(graphene.ObjectType):
    all_ingredients = graphene.List(IngredientType)
    category_by_name = graphene.Field(CategoryType, name=graphene.
↪String(required=True))

    def resolve_all_ingredients(root, info):
        # We can easily optimize query count in the resolve method
        return Ingredient.objects.select_related("category").all()

    def resolve_category_by_name(root, info, name):
        try:
            return Category.objects.get(name=name)
        except Category.DoesNotExist:
            return None

schema = graphene.Schema(query=Query)
```

You can think of this as being something like your top-level `urls.py` file.

1.2.3 Testing everything so far

We are going to do some configuration work, in order to have a working Django where we can test queries, before we move on, updating our schema.

Update settings

Next, install your app and GraphiQL in your Django project. GraphiQL is a web-based integrated development environment to assist in the writing and executing of GraphQL queries. It will provide us with a simple and easy way

of testing our cookbook project.

Add `graphene_django` to `INSTALLED_APPS` in `cookbook/settings.py`:

```
# cookbook/settings.py

INSTALLED_APPS = [
    ...
    "graphene_django",
]
```

And then add the `SCHEMA` to the `GRAPHENE` config in `cookbook/settings.py`:

```
# cookbook/settings.py

GRAPHENE = {
    "SCHEMA": "cookbook.schema.schema"
}
```

Alternatively, we can specify the schema to be used in the urls definition, as explained below.

Creating GraphQL and GraphiQL views

Unlike a RESTful API, there is only a single URL from which GraphQL is accessed. Requests to this URL are handled by Graphene's `GraphQLView` view.

This view will serve as GraphQL endpoint. As we want to have the aforementioned GraphiQL we specify that on the parameters with `graphiql=True`.

```
# cookbook/urls.py

from django.contrib import admin
from django.urls import path
from django.views.decorators.csrf import csrf_exempt

from graphene_django.views import GraphQLView

urlpatterns = [
    path("admin/", admin.site.urls),
    path("graphql", csrf_exempt(GraphQLView.as_view(graphiql=True))),
]
```

If we didn't specify the target schema in the Django settings file as explained above, we can do so here using:

```
# cookbook/urls.py

from django.contrib import admin
from django.urls import path
from django.views.decorators.csrf import csrf_exempt

from graphene_django.views import GraphQLView

from cookbook.schema import schema

urlpatterns = [
    path("admin/", admin.site.urls),
    path("graphql", csrf_exempt(GraphQLView.as_view(graphiql=True, schema=schema))),
]
```

Testing our GraphQL schema

We're now ready to test the API we've built. Let's fire up the server from the command line.

```
python manage.py runserver

Performing system checks...
Django version 3.0.7, using settings 'cookbook.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Go to localhost:8000/graphql and type your first query!

```
query {
  allIngredients {
    id
    name
  }
}
```

If you are using the provided fixtures, you will see the following response:

```
{
  "data": {
    "allIngredients": [
      {
        "id": "1",
        "name": "Eggs"
      },
      {
        "id": "2",
        "name": "Milk"
      },
      {
        "id": "3",
        "name": "Beef"
      },
      {
        "id": "4",
        "name": "Chicken"
      }
    ]
  }
}
```

Congratulations, you have created a working GraphQL server !

Note: Graphene automatically camelcases all field names for better compatibility with JavaScript clients.

1.2.4 Getting relations

Using the current schema we can query for relations too. This is where GraphQL becomes really powerful!

For example, we may want to get a specific categories and list all ingredients that are in that category.

We can do that with the following query:

```
query {
  categoryByName(name: "Dairy") {
    id
    name
    ingredients {
      id
      name
    }
  }
}
```

This will give you (in case you are using the fixtures) the following result:

```
{
  "data": {
    "categoryByName": {
      "id": "1",
      "name": "Dairy",
      "ingredients": [
        {
          "id": "1",
          "name": "Eggs"
        },
        {
          "id": "2",
          "name": "Milk"
        }
      ]
    }
  }
}
```

We can also list all ingredients and get information for the category they are in:

```
query {
  allIngredients {
    id
    name
    category {
      id
      name
    }
  }
}
```

1.2.5 Summary

As you can see, GraphQL is very powerful and integrating Django models allows you to get started with a working server quickly.

If you want to put things like `django-filter` and automatic pagination in action, you should continue with the *Relay tutorial*.

A good idea is to check the Graphene documentation so that you are familiar with it as well.

1.3 Relay tutorial

Graphene has a number of additional features that are designed to make working with Django *really simple*.

Note: The code in this quickstart is pulled from the cookbook example app.

A good idea is to check the following things first:

- Graphene Relay documentation
- GraphQL Relay Specification

1.3.1 Setup the Django project

We will setup the project, create the following:

- A Django project called `cookbook`
- An app within `cookbook` called `ingredients`

```
# Create the project directory
mkdir cookbook
cd cookbook

# Create a virtualenv to isolate our package dependencies locally
virtualenv env
source env/bin/activate  # On Windows use `env\Scripts\activate`

# Install Django and Graphene with Django support
pip install django
pip install graphene_django

# Set up a new project with a single application
django-admin.py startproject cookbook .  # Note the trailing '.' character
cd cookbook
django-admin.py startapp ingredients
```

Now sync your database for the first time:

```
python manage.py migrate
```

Let's create a few simple models...

Defining our models

Let's get started with these models:

```
# cookbook/ingredients/models.py
from django.db import models


class Category(models.Model):
    name = models.CharField(max_length=100)

    def __str__(self):
        return self.name
```

```
class Ingredient(models.Model):
    name = models.CharField(max_length=100)
    notes = models.TextField()
    category = models.ForeignKey(Category, related_name='ingredients')

    def __str__(self):
        return self.name
```

Don't forget to create & run migrations:

```
python manage.py makemigrations
python manage.py migrate
```

Load some test data

Now is a good time to load up some test data. The easiest option will be to download the ingredients.json fixture and place it in `cookbook/ingredients/fixtures/ingredients.json`. You can then run the following:

```
$ python ./manage.py loaddata ingredients

Installed 6 object(s) from 1 fixture(s)
```

Alternatively you can use the Django admin interface to create some data yourself. You'll need to run the development server (see below), and create a login for yourself too (`./manage.py createsuperuser`).

1.3.2 Schema

GraphQL presents your objects to the world as a graph structure rather than a more hierarchical structure to which you may be accustomed. In order to create this representation, Graphene needs to know about each *type* of object which will appear in the graph.

This graph also has a *root type* through which all access begins. This is the `Query` class below. In this example, we provide the ability to list all ingredients via `all_ingredients`, and the ability to obtain a specific ingredient via `ingredient`.

Create `cookbook/ingredients/schema.py` and type the following:

```
# cookbook/ingredients/schema.py
from graphene import relay, ObjectType
from graphene_django import DjangoObjectType
from graphene_django.filter import DjangoFilterConnectionField

from ingredients.models import Category, Ingredient


# Graphene will automatically map the Category model's fields onto the CategoryNode.
# This is configured in the CategoryNode's Meta class (as you can see below)
class CategoryNode(DjangoObjectType):
    class Meta:
        model = Category
        filter_fields = ['name', 'ingredients']
        interfaces = (relay.Node, )
```

```
class IngredientNode(DjangoObjectType):
    class Meta:
        model = Ingredient
        # Allow for some more advanced filtering here
        filter_fields = {
            'name': ['exact', 'icontains', 'istartswith'],
            'notes': ['exact', 'icontains'],
            'category': ['exact'],
            'category__name': ['exact'],
        }
        interfaces = (relay.Node, )


class Query(graphene.ObjectType):
    category = relay.Node.Field(CategoryNode)
    all_categories = DjangoFilterConnectionField(CategoryNode)

    ingredient = relay.Node.Field(IngredientNode)
    all_ingredients = DjangoFilterConnectionField(IngredientNode)
```

The filtering functionality is provided by django-filter. See the usage documentation for details on the format for `filter_fields`. While optional, this tutorial makes use of this functionality so you will need to install `django-filter` for this tutorial to work:

```
pip install django-filter
```

Note that the above `Query` class is marked as 'abstract'. This is because we will now create a project-level query which will combine all our app-level queries.

Create the parent project-level `cookbook/schema.py`:

```
import graphene

import ingredients.schema


class Query(ingredients.schema.Query, graphene.ObjectType):
    # This class will inherit from multiple Queries
    # as we begin to add more apps to our project
    pass

schema = graphene.Schema(query=Query)
```

You can think of this as being something like your top-level `urls.py` file (although it currently lacks any namespacing).

1.3.3 Testing everything so far

Update settings

Next, install your app and GraphiQL in your Django project. GraphiQL is a web-based integrated development environment to assist in the writing and executing of GraphQL queries. It will provide us with a simple and easy way of testing our cookbook project.

Add `ingredients` and `graphene_django` to `INSTALLED_APPS` in `cookbook/settings.py`:

```
INSTALLED_APPS = [
    ...
    # This will also make the `graphql_schema` management command available
    'graphene_django',

    # Install the ingredients app
    'ingredients',
]
```

And then add the `SCHEMA` to the `GRAPHENE` config in `cookbook/settings.py`:

```
GRAPHENE = {
    'SCHEMA': 'cookbook.schema.schema'
}
```

Alternatively, we can specify the schema to be used in the urls definition, as explained below.

Creating GraphQL and GraphiQL views

Unlike a RESTful API, there is only a single URL from which GraphQL is accessed. Requests to this URL are handled by Graphene's `GraphQLView` view.

This view will serve as GraphQL endpoint. As we want to have the aforementioned GraphiQL we specify that on the params with `graphiql=True`.

```
from django.conf.urls import url, include
from django.contrib import admin

from graphene_django.views import GraphQLView

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^graphql$', GraphQLView.as_view(graphiql=True)),
]
```

If we didn't specify the target schema in the Django settings file as explained above, we can do so here using:

```
from django.conf.urls import url, include
from django.contrib import admin

from graphene_django.views import GraphQLView

from cookbook.schema import schema

urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^graphql$', GraphQLView.as_view(graphiql=True, schema=schema)),
]
```

Testing our GraphQL schema

We're now ready to test the API we've built. Let's fire up the server from the command line.

```
$ python ./manage.py runserver
```

```
Performing system checks...
Django version 1.11, using settings 'cookbook.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Go to localhost:8000/graphql and type your first query!

```
query {
  allIngredients {
    edges {
      node {
        id,
        name
      }
    }
  }
}
```

The above will return the names & IDs for all ingredients. But perhaps you want a specific ingredient:

```
query {
  # Graphene creates globally unique IDs for all objects.
  # You may need to copy this value from the results of the first query
  ingredient(id: "SW5ncmVkaWVudE5vZGU6MQ==") {
    name
  }
}
```

You can also get each ingredient for each category:

```
query {
  allCategories {
    edges {
      node {
        name,
        ingredients {
          edges {
            node {
              name
            }
          }
        }
      }
    }
  }
}
```

Or you can get only 'meat' ingredients containing the letter 'e':

```
query {
  # You can also use `category: "CATEGORY GLOBAL ID"`
  allIngredients(name_Icontains: "e", category_Name: "Meat") {
    edges {
      node {
        name
      }
    }
```

```
  }
}
```

Final Steps

We have created a GraphQL endpoint that will work with Relay, but for Relay to work it needs access to a (non python) schema. Instructions to export the schema can be found on the Introspection Schema part of this guide.

1.4 Schema

The `graphene.Schema` object describes your data model and provides a GraphQL server with an associated set of resolve methods that know how to fetch data. The most basic schema you can create looks like this:

```
import graphene

class Query(graphene.ObjectType):
    pass

class Mutation(graphene.ObjectType):
    pass

schema = graphene.Schema(query=Query, mutation=Mutation)
```

This schema doesn't do anything yet, but it is ready to accept new Query or Mutation fields.

1.4.1 Adding to the schema

If you have defined a `Query` or `Mutation`, you can register them with the schema:

```
import graphene

import my_app.schema.Query
import my_app.schema.Mutation

class Query(
    my_app.schema.Query, # Add your Query objects here
    graphene.ObjectType
):
    pass

class Mutation(
    my_app.schema.Mutation, # Add your Mutation objects here
    graphene.ObjectType
):
    pass

schema = graphene.Schema(query=Query, mutation=Mutation)
```

You can add as many mixins to the base `Query` and `Mutation` objects as you like.

Read more about Schema on the core graphene docs

1.5 Queries & ObjectTypes

1.5.1 Introduction

Graphene-Django offers a host of features for performing GraphQL queries.

Graphene-Django ships with a special `DjangoObjectType` that automatically transforms a Django Model into a `ObjectType` for you.

Full example

```
# my_app/schema.py

import graphene
from graphene_django import DjangoObjectType

from .models import Question

class QuestionType(DjangoObjectType):
    class Meta:
        model = Question
        fields = ("id", "question_text")

class Query(graphene.ObjectType):
    questions = graphene.List(QuestionType)
    question_by_id = graphene.Field(QuestionType, id=graphene.String())

    def resolve_questions(root, info, **kwargs):
        # Querying a list
        return Question.objects.all()

    def resolve_question_by_id(root, info, id):
        # Querying a single question
        return Question.objects.get(pk=id)
```

1.5.2 Specifying which fields to include

By default, `DjangoObjectType` will present all fields on a Model through GraphQL. If you only want a subset of fields to be present, you can do so using `fields` or `exclude`. It is strongly recommended that you explicitly set all fields that should be exposed using the fields attribute. This will make it less likely to result in unintentionally exposing data when your models change.

`fields`

Show **only** these fields on the model:

```
from graphene_django import DjangoObjectType
from .models import Question

class QuestionType(DjangoObjectType):
    class Meta:
        model = Question
        fields = ("id", "question_text")
```

You can also set the `fields` attribute to the special value `"__all__"` to indicate that all fields in the model should be used.

For example:

```
from graphene_django import DjangoObjectType
from .models import Question

class QuestionType(DjangoObjectType):
    class Meta:
        model = Question
        fields = "__all__"
```

exclude

Show all fields **except** those in `exclude`:

```
from graphene_django import DjangoObjectType
from .models import Question

class QuestionType(DjangoObjectType):
    class Meta:
        model = Question
        exclude = ("question_text",)
```

1.5.3 Customising fields

You can completely overwrite a field, or add new fields, to a `DjangoObjectType` using a Resolver:

```
from graphene_django import DjangoObjectType
from .models import Question

class QuestionType(DjangoObjectType):

    class Meta:
        model = Question
        fields = ("id", "question_text")

    extra_field = graphene.String()

    def resolve_extra_field(self, info):
        return "hello!"
```

Choices to Enum conversion

By default Graphene-Django will convert any Django fields that have choices defined into a GraphQL enum type.

For example the following `Model` and `DjangoObjectType`:

```
from django.db import models
from graphene_django import DjangoObjectType

class PetModel(models.Model):
    kind = models.CharField(
```

```
        max_length=100,
        choices=(("cat", "Cat"), ("dog", "Dog"))
    )

class Pet(DjangoObjectType):
    class Meta:
        model = PetModel
        fields = ("id", "kind",)
```

Results in the following GraphQL schema definition:

```
type Pet {
  id: ID!
  kind: PetModelKind!
}

enum PetModelKind {
  CAT
  DOG
}
```

You can disable this automatic conversion by setting `convert_choices_to_enum` attribute to `False` on the `DjangoObjectType` `Meta` class.

```
from graphene_django import DjangoObjectType
from .models import PetModel

class Pet(DjangoObjectType):
    class Meta:
        model = PetModel
        fields = ("id", "kind",)
        convert_choices_to_enum = False
```

```
type Pet {
  id: ID!
  kind: String!
}
```

You can also set `convert_choices_to_enum` to a list of fields that should be automatically converted into enums:

```
from graphene_django import DjangoObjectType
from .models import PetModel

class Pet(DjangoObjectType):
    class Meta:
        model = PetModel
        fields = ("id", "kind",)
        convert_choices_to_enum = ["kind"]
```

Note: Setting `convert_choices_to_enum = []` is the same as setting it to `False`.

1.5.4 Related models

Say you have the following models:

```
from django.db import models

class Category(models.Model):
    foo = models.CharField(max_length=256)

class Question(models.Model):
    category = models.ForeignKey(Category, on_delete=models.CASCADE)
```

When `Question` is published as a `DjangoObjectType` and you want to add `Category` as a query-able field like so:

```
from graphene_django import DjangoObjectType
from .models import Question

class QuestionType(DjangoObjectType):
    class Meta:
        model = Question
        fields = ("category",)
```

Then all query-able related models must be defined as DjangoObjectType subclass, or they will fail to show if you are trying to query those relation fields. You only need to create the most basic class for this to work:

```
from graphene_django import DjangoObjectType
from .models import Category

class CategoryType(DjangoObjectType):
    class Meta:
        model = Category
        fields = ("foo",)
```

1.5.5 Default QuerySet

If you are using `DjangoObjectType` you can define a custom *get_queryset* method. Use this to control filtering on the ObjectType level instead of the Query object level.

```
from graphene_django.types import DjangoObjectType
from .models import Question

class QuestionType(DjangoObjectType):
    class Meta:
        model = Question

    @classmethod
    def get_queryset(cls, queryset, info):
        if info.context.user.is_anonymous:
            return queryset.filter(published=True)
        return queryset
```

1.5.6 Resolvers

When a GraphQL query is received by the `Schema` object, it will map it to a "Resolver" related to it.

This resolve method should follow this format:

```
def resolve_foo(parent, info, **kwargs):
```

Where "foo" is the name of the field declared in the `Query` object.

```
import graphene
from .models import Question
from .types import QuestionType

class Query(graphene.ObjectType):
    foo = graphene.List(QuestionType)

    def resolve_foo(root, info, **kwargs):
        id = kwargs.get("id")
        return Question.objects.get(id)
```

Arguments

Additionally, Resolvers will receive **any arguments declared in the field definition**. This allows you to provide input arguments in your GraphQL server and can be useful for custom queries.

```
import graphene
from .models import Question
from .types import QuestionType

class Query(graphene.ObjectType):
    question = graphene.Field(
        QuestionType,
        foo=graphene.String(),
        bar=graphene.Int()
    )

    def resolve_question(root, info, foo, bar):
        # If `foo` or `bar` are declared in the GraphQL query they will be here, else␣
↪None.
        return Question.objects.filter(foo=foo, bar=bar).first()
```

Info

The `info` argument passed to all resolve methods holds some useful information. For Graphene-Django, the `info.context` attribute is the `HTTPRequest` object that would be familiar to any Django developer. This gives you the full functionality of Django's `HTTPRequest` in your resolve methods, such as checking for authenticated users:

```
import graphene

from .models import Question
from .types import QuestionType

class Query(graphene.ObjectType):
    questions = graphene.List(QuestionType)

def resolve_questions(root, info):
    # See if a user is authenticated
    if info.context.user.is_authenticated():
        return Question.objects.all()
```

```
    else:
        return Question.objects.none()
```

DjangoObjectTypes

A Resolver that maps to a defined *DjangoObjectType* should only use methods that return a queryset. Queryset methods like *values* will return dictionaries, use *defer* instead.

1.5.7 Plain ObjectTypes

With Graphene-Django you are not limited to just Django Models - you can use the standard `ObjectType` to create custom fields or to provide an abstraction between your internal Django models and your external API.

```
import graphene
from .models import Question

class MyQuestion(graphene.ObjectType):
    text = graphene.String()

class Query(graphene.ObjectType):
    question = graphene.Field(MyQuestion, question_id=graphene.String())

    def resolve_question(root, info, question_id):
        question = Question.objects.get(pk=question_id)
        return MyQuestion(
            text=question.question_text
        )
```

For more information and more examples, please see the core object type documentation.

1.5.8 Relay

Relay with Graphene-Django gives us some additional features:

- Pagination and slicing.
- An abstract `id` value which contains enough info for the server to know its type and its id.

There is one additional import and a single line of code needed to adopt this:

Full example

See the Relay documentation on the core graphene pages for more information on customizing the Relay experience.

```
from graphene import relay
from graphene_django import DjangoObjectType
from .models import Question

class QuestionType(DjangoObjectType):
    class Meta:
        model = Question
        interfaces = (relay.Node,)  # make sure you add this
        fields = "__all__"
```

```
class QuestionConnection(relay.Connection):
    class Meta:
        node = QuestionType

class Query:
    questions = relay.ConnectionField(QuestionConnection)

    def resolve_questions(root, info, **kwargs):
        return Question.objects.all()
```

You can now execute queries like:

```
{
    questions (first: 2, after: "YXJyYXljb25uZWN0aW9uOjEwNQ==") {
        pageInfo {
        startCursor
        endCursor
        hasNextPage
        hasPreviousPage
        }
        edges {
        cursor
        node {
            id
            question_text
        }
        }
    }
}
```

Which returns:

```
{
    "data": {
        "questions": {
        "pageInfo": {
            "startCursor": "YXJyYXljb25uZWN0aW9uOjEwNg==",
            "endCursor": "YXJyYXljb25uZWN0aW9uOjEwNw==",
            "hasNextPage": true,
            "hasPreviousPage": false
        },
        "edges": [
            {
            "cursor": "YXJyYXljb25uZWN0aW9uOjEwNg==",
            "node": {
                "id": "UGxhY2VUeXBlOjEwNw==",
                "question_text": "How did we get here?"
            }
            },
            {
            "cursor": "YXJyYXljb25uZWN0aW9uOjEwNw==",
            "node": {
                "id": "UGxhY2VUeXBlOjEwOA==",
                "name": "Where are we?"
            }
            }
        ]
        }
```

```
    }
}
```

Note that relay implements `pagination` capabilities automatically, adding a `pageInfo` element, and including `cursor` on nodes. These elements are included in the above example for illustration.

To learn more about Pagination in general, take a look at Pagination on the GraphQL community site.

1.6 Fields

Graphene-Django provides some useful fields to help integrate Django with your GraphQL Schema.

1.6.1 DjangoListField

`DjangoListField` allows you to define a list of *DjangoObjectType*'s. By default it will resolve the default queryset of the Django model.

```
from graphene import ObjectType, Schema
from graphene_django import DjangoListField

class RecipeType(DjangoObjectType):
   class Meta:
      model = Recipe
      fields = ("title", "instructions")

class Query(ObjectType):
   recipes = DjangoListField(RecipeType)

schema = Schema(query=Query)
```

The above code results in the following schema definition:

```
schema {
  query: Query
}

type Query {
  recipes: [RecipeType!]
}

type RecipeType {
  title: String!
  instructions: String!
}
```

Custom resolvers

If your `DjangoObjectType` has defined a custom *get_queryset* method, when resolving a `DjangoListField` it will be called with either the return of the field resolver (if one is defined) or the default queryset from the Django model.

For example the following schema will only resolve recipes which have been published and have a title:

```
from graphene import ObjectType, Schema
from graphene_django import DjangoListField

class RecipeType(DjangoObjectType):
   class Meta:
      model = Recipe
      fields = ("title", "instructions")

   @classmethod
   def get_queryset(cls, queryset, info):
      # Filter out recipes that have no title
      return queryset.exclude(title__exact="")

class Query(ObjectType):
   recipes = DjangoListField(RecipeType)

   def resolve_recipes(parent, info):
      # Only get recipes that have been published
      return Recipe.objects.filter(published=True)

schema = Schema(query=Query)
```

1.6.2 DjangoConnectionField

TODO

1.7 Extra Types

Here are some libraries that provide common types for Django specific fields.

1.7.1 GeoDjango

Use the graphene-gis library to add GeoDjango types to your Schema.

1.8 Mutations

1.8.1 Introduction

Graphene-Django makes it easy to perform mutations.

With Graphene-Django we can take advantage of pre-existing Django features to quickly build CRUD functionality, while still using the core graphene mutation features to add custom mutations to a Django project.

1.8.2 Simple example

```
import graphene

from graphene_django import DjangoObjectType
```

```
from .models import Question


class QuestionType(DjangoObjectType):
    class Meta:
        model = Question


class QuestionMutation(graphene.Mutation):
    class Arguments:
        # The input arguments for this mutation
        text = graphene.String(required=True)
        id = graphene.ID()

    # The class attributes define the response of the mutation
    question = graphene.Field(QuestionType)

    @classmethod
    def mutate(cls, root, info, text, id):
        question = Question.objects.get(pk=id)
        question.text = text
        question.save()
        # Notice we return an instance of this mutation
        return QuestionMutation(question=question)


class Mutation(graphene.ObjectType):
    update_question = QuestionMutation.Field()
```

1.8.3 Django Forms

Graphene-Django comes with mutation classes that will convert the fields on Django forms into inputs on a mutation.

DjangoFormMutation

```
from graphene_django.forms.mutation import DjangoFormMutation

class MyForm(forms.Form):
    name = forms.CharField()

class MyMutation(DjangoFormMutation):
    class Meta:
        form_class = MyForm
```

`MyMutation` will automatically receive an `input` argument. This argument should be a `dict` where the key is `name` and the value is a string.

DjangoModelFormMutation

`DjangoModelFormMutation` will pull the fields from a `ModelForm`.

```
from graphene_django.forms.mutation import DjangoModelFormMutation

class Pet(models.Model):
```

```
    name = models.CharField()

class PetForm(forms.ModelForm):
    class Meta:
        model = Pet
        fields = ('name',)

# This will get returned when the mutation completes successfully
class PetType(DjangoObjectType):
    class Meta:
        model = Pet

class PetMutation(DjangoModelFormMutation):
    pet = Field(PetType)

    class Meta:
        form_class = PetForm
```

`PetMutation` will grab the fields from `PetForm` and turn them into inputs. If the form is valid then the mutation will lookup the `DjangoObjectType` for the `Pet` model and return that under the key `pet`. Otherwise it will return a list of errors.

You can change the input name (default is `input`) and the return field name (default is the model name lowercase).

```
class PetMutation(DjangoModelFormMutation):
    class Meta:
        form_class = PetForm
        input_field_name = 'data'
        return_field_name = 'my_pet'
```

Form validation

Form mutations will call `is_valid()` on your forms.

If the form is valid then the class method `perform_mutate(form, info)` is called on the mutation. Override this method to change how the form is saved or to return a different Graphene object type.

If the form is *not* valid then a list of errors will be returned. These errors have two fields: `field`, a string containing the name of the invalid form field, and `messages`, a list of strings with the validation messages.

1.8.4 Django REST Framework

You can re-use your Django Rest Framework serializer with Graphene Django mutations.

You can create a Mutation based on a serializer by using the *SerializerMutation* base class:

```
from graphene_django.rest_framework.mutation import SerializerMutation

class MyAwesomeMutation(SerializerMutation):
    class Meta:
        serializer_class = MySerializer
```

Create/Update Operations

By default ModelSerializers accept create and update operations. To customize this use the *model_operations* attribute on the `SerializerMutation` class.

The update operation looks up models by the primary key by default. You can customize the look up with the `lookup_field` attribute on the `SerializerMutation` class.

```
from graphene_django.rest_framework.mutation import SerializerMutation
from .serializers import MyModelSerializer


class AwesomeModelMutation(SerializerMutation):
    class Meta:
        serializer_class = MyModelSerializer
        model_operations = ['create', 'update']
        lookup_field = 'id'
```

Overriding Update Queries

Use the method `get_serializer_kwargs` to override how updates are applied.

```
from graphene_django.rest_framework.mutation import SerializerMutation
from .serializers import MyModelSerializer


class AwesomeModelMutation(SerializerMutation):
    class Meta:
        serializer_class = MyModelSerializer

    @classmethod
    def get_serializer_kwargs(cls, root, info, **input):
        if 'id' in input:
            instance = Post.objects.filter(
                id=input['id'], owner=info.context.user
            ).first()
            if instance:
                return {'instance': instance, 'data': input, 'partial': True}

            else:
                raise http.Http404

        return {'data': input, 'partial': True}
```

1.8.5 Relay

You can use relay with mutations. A Relay mutation must inherit from `ClientIDMutation` and implement the `mutate_and_get_payload` method:

```
import graphene
from graphene import relay
from graphene_django import DjangoObjectType
from graphql_relay import from_global_id

from .queries import QuestionType
```

```
class QuestionMutation(relay.ClientIDMutation):
    class Input:
        text = graphene.String(required=True)
        id = graphene.ID()

    question = graphene.Field(QuestionType)

    @classmethod
    def mutate_and_get_payload(cls, root, info, text, id):
        question = Question.objects.get(pk=from_global_id(id)[1])
        question.text = text
        question.save()
        return QuestionMutation(question=question)
```

Notice that the `class Arguments` is renamed to `class Input` with relay. This is due to a deprecation of `class Arguments` in graphene 2.0.

Relay ClientIDMutation accept a `clientIDMutation` argument. This argument is also sent back to the client with the mutation result (you do not have to do anything). For services that manage a pool of many GraphQL requests in bulk, the `clientIDMutation` allows you to match up a specific mutation with the response.

1.8.6 Django Database Transactions

Django gives you a few ways to control how database transactions are managed.

Tying transactions to HTTP requests

A common way to handle transactions in Django is to wrap each request in a transaction. Set `ATOMIC_REQUESTS` settings to `True` in the configuration of each database for which you want to enable this behavior.

It works like this. Before calling `GraphQLView` Django starts a transaction. If the response is produced without problems, Django commits the transaction. If the view, a `DjangoFormMutation` or a `DjangoModelFormMutation` produces an exception, Django rolls back the transaction.

> **Warning:** While the simplicity of this transaction model is appealing, it also makes it inefficient when traffic increases. Opening a transaction for every request has some overhead. The impact on performance depends on the query patterns of your application and on how well your database handles locking.

Check the next section for a better solution.

Tying transactions to mutations

A mutation can contain multiple fields, just like a query. There's one important distinction between queries and mutations, other than the name:

> *While query fields are executed in parallel, mutation fields run in series, one after the other.*

This means that if we send two `incrementCredits` mutations in one request, the first is guaranteed to finish before the second begins, ensuring that we don't end up with a race condition with ourselves.

On the other hand, if the first `incrementCredits` runs successfully but the second one does not, the operation cannot be retried as it is. That's why is a good idea to run all mutation fields in a transaction, to guarantee all occur or nothing occurs.

To enable this behavior for all databases set the graphene `ATOMIC_MUTATIONS` settings to `True` in your settings file:

```
GRAPHENE = {
    # ...
    "ATOMIC_MUTATIONS": True,
}
```

On the contrary, if you want to enable this behavior for a specific database, set `ATOMIC_MUTATIONS` to `True` in your database settings:

```
DATABASES = {
    "default": {
        # ...
        "ATOMIC_MUTATIONS": True,
    },
    # ...
}
```

Now, given the following example mutation:

```
mutation IncreaseCreditsTwice {

    increaseCredits1: increaseCredits(input: { amount: 10 }) {
        balance
        errors {
            field
            messages
        }
    }

    increaseCredits2: increaseCredits(input: { amount: -1 }) {
        balance
        errors {
            field
            messages
        }
    }

}
```

The server is going to return something like:

```
{
    "data": {
        "increaseCredits1": {
            "balance": 10.0,
            "errors": []
        },
        "increaseCredits2": {
            "balance": null,
            "errors": [
                {
                    "field": "amount",
                    "message": "Amount should be a positive number"
```

```
                    }
                ]
            },
        }
}
```

But the balance will remain the same.

1.9 Subscriptions

The `graphene-django` project does not currently support GraphQL subscriptions out of the box. However, there are several community-driven modules for adding subscription support, and the provided GraphiQL interface supports running subscription operations over a websocket.

To implement websocket-based support for GraphQL subscriptions, you'll need to do the following:

1. Install and configure django-channels.
2. Install and configure* a third-party module for adding subscription support over websockets. A few options include:
   - graphql-python/graphql-ws
   - datavance/django-channels-graphql-ws
   - jaydenwindle/graphene-subscriptions
3. Ensure that your application (or at least your GraphQL endpoint) is being served via an ASGI protocol server like daphne (built in to `django-channels`), uvicorn, or hypercorn.

   *** Note:** By default, the GraphiQL interface that comes with `graphene-django` assumes that you are handling subscriptions at the same path as any other operation (i.e., you configured both `urls.py` and `routing.py` to handle GraphQL operations at the same path, like `/graphql`).

   If these URLs differ, GraphiQL will try to run your subscription over HTTP, which will produce an error. If you need to use a different URL for handling websocket connections, you can configure `SUBSCRIPTION_PATH` in your `settings.py`:

```
GRAPHENE = {
    # ...
    "SUBSCRIPTION_PATH": "/ws/graphql"  # The path you configured in␣
↪`routing.py`, including a leading slash.
}
```

Once your application is properly configured to handle subscriptions, you can use the GraphiQL interface to test subscriptions like any other operation.

1.10 Filtering

Graphene-Django integrates with django-filter (2.x for Python 3 or 1.x for Python 2) to provide filtering of results. See the usage documentation for details on the format for `filter_fields`.

This filtering is automatically available when implementing a `relay.Node`. Additionally `django-filter` is an optional dependency of Graphene.

You will need to install it manually, which can be done as follows:

```
# You'll need to install django-filter
pip install django-filter>=2
```

After installing `django-filter` you'll need to add the application in the `settings.py` file:

```
INSTALLED_APPS = [
    # ...
    "django_filters",
]
```

Note: The techniques below are demoed in the cookbook example app.

1.10.1 Filterable fields

The `filter_fields` parameter is used to specify the fields which can be filtered upon. The value specified here is passed directly to `django-filter`, so see the filtering documentation for full details on the range of options available.

For example:

```
class AnimalNode(DjangoObjectType):
    class Meta:
        # Assume you have an Animal model defined with the following fields
        model = Animal
        filter_fields = ['name', 'genus', 'is_domesticated']
        interfaces = (relay.Node, )

class Query(ObjectType):
    animal = relay.Node.Field(AnimalNode)
    all_animals = DjangoFilterConnectionField(AnimalNode)
```

You could then perform a query such as:

```
query {
  # Note that fields names become camelcased
  allAnimals(genus: "cat", isDomesticated: true) {
    edges {
      node {
        id,
        name
      }
    }
  }
}
```

You can also make more complex lookup types available:

```
class AnimalNode(DjangoObjectType):
    class Meta:
        model = Animal
        # Provide more complex lookup types
        filter_fields = {
            'name': ['exact', 'icontains', 'istartswith'],
            'genus': ['exact'],
            'is_domesticated': ['exact'],
        }
        interfaces = (relay.Node, )
```

Which you could query as follows:

```
query {
  # Note that fields names become camelcased
  allAnimals(name_Icontains: "lion") {
    edges {
      node {
        id,
        name
      }
    }
  }
}
```

1.10.2 Custom Filtersets

By default Graphene provides easy access to the most commonly used features of `django-filter`. This is done by transparently creating a `django_filters.FilterSet` class for you and passing in the values for `filter_fields`.

However, you may find this to be insufficient. In these cases you can create your own `FilterSet`. You can pass it directly as follows:

```
class AnimalNode(DjangoObjectType):
    class Meta:
        # Assume you have an Animal model defined with the following fields
        model = Animal
        filter_fields = ['name', 'genus', 'is_domesticated']
        interfaces = (relay.Node, )


class AnimalFilter(django_filters.FilterSet):
    # Do case-insensitive lookups on 'name'
    name = django_filters.CharFilter(lookup_expr=['iexact'])

    class Meta:
        model = Animal
        fields = ['name', 'genus', 'is_domesticated']


class Query(ObjectType):
    animal = relay.Node.Field(AnimalNode)
    # We specify our custom AnimalFilter using the filterset_class param
    all_animals = DjangoFilterConnectionField(AnimalNode,
                                              filterset_class=AnimalFilter)
```

You can also specify the `FilterSet` class using the `filterset_class` parameter when defining your `DjangoObjectType`, however, this can't be used in unison with the `filter_fields` parameter:

```
class AnimalFilter(django_filters.FilterSet):
    # Do case-insensitive lookups on 'name'
    name = django_filters.CharFilter(lookup_expr=['iexact'])

    class Meta:
        # Assume you have an Animal model defined with the following fields
        model = Animal
        fields = ['name', 'genus', 'is_domesticated']
```

```
class AnimalNode(DjangoObjectType):
    class Meta:
        model = Animal
        filterset_class = AnimalFilter
        interfaces = (relay.Node, )


class Query(ObjectType):
    animal = relay.Node.Field(AnimalNode)
    all_animals = DjangoFilterConnectionField(AnimalNode)
```

The context argument is passed on as the request argument in a `django_filters.FilterSet` instance. You can use this to customize your filters to be context-dependent. We could modify the `AnimalFilter` above to pre-filter animals owned by the authenticated user (set in `context.user`).

```
class AnimalFilter(django_filters.FilterSet):
    # Do case-insensitive lookups on 'name'
    name = django_filters.CharFilter(lookup_type=['iexact'])

    class Meta:
        model = Animal
        fields = ['name', 'genus', 'is_domesticated']

    @property
    def qs(self):
        # The query context can be found in self.request.
        return super(AnimalFilter, self).qs.filter(owner=self.request.user)
```

1.10.3 Ordering

You can use `OrderFilter` to define how you want your returned results to be ordered.

Extend the tuple of fields if you want to order by more than one field.

```
from django_filters import FilterSet, OrderingFilter

class UserFilter(FilterSet):
    class Meta:
        model = UserModel

    order_by = OrderingFilter(
        fields=(
            ('name', 'created_at'),
        )
    )

class Group(DjangoObjectType):
  users = DjangoFilterConnectionField(Ticket, filterset_class=UserFilter)

  class Meta:
      name = 'Group'
      model = GroupModel
      interfaces = (relay.Node,)
```

```
  def resolve_users(self, info, **kwargs):
    return UserFilter(kwargs).qs
```

with this set up, you can now order the users under group:

```
query {
  group(id: "xxx") {
    users(orderBy: "-created_at") {
      xxx
    }
  }
}
```

1.11 Authorization in Django

There are several ways you may want to limit access to data when working with Graphene and Django: limiting which fields are accessible via GraphQL and limiting which objects a user can access.

Let's use a simple example model.

```
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    published = models.BooleanField(default=False)
    owner = models.ForeignKey('auth.User')
```

1.11.1 Limiting Field Access

To limit fields in a GraphQL query simply use the `fields` meta attribute.

```
from graphene import relay
from graphene_django.types import DjangoObjectType
from .models import Post

class PostNode(DjangoObjectType):
    class Meta:
        model = Post
        fields = ('title', 'content')
        interfaces = (relay.Node, )
```

conversely you can use `exclude` meta attribute.

```
from graphene import relay
from graphene_django.types import DjangoObjectType
from .models import Post

class PostNode(DjangoObjectType):
    class Meta:
        model = Post
        exclude = ('published', 'owner')
        interfaces = (relay.Node, )
```

1.11.2 Queryset Filtering On Lists

In order to filter which objects are available in a queryset-based list, define a resolve method for that field and return the desired queryset.

```
from graphene import ObjectType
from graphene_django.filter import DjangoFilterConnectionField
from .models import Post

class Query(ObjectType):
    all_posts = DjangoFilterConnectionField(PostNode)

    def resolve_all_posts(self, info):
         return Post.objects.filter(published=True)
```

1.11.3 User-based Queryset Filtering

If you are using `GraphQLView` you can access Django's request with the context argument.

```
from graphene import ObjectType
from graphene_django.filter import DjangoFilterConnectionField
from .models import Post

class Query(ObjectType):
    my_posts = DjangoFilterConnectionField(PostNode)

    def resolve_my_posts(self, info):
        # context will reference to the Django request
        if not info.context.user.is_authenticated:
            return Post.objects.none()
        else:
            return Post.objects.filter(owner=info.context.user)
```

If you're using your own view, passing the request context into the schema is simple.

```
result = schema.execute(query, context_value=request)
```

1.11.4 Global Filtering

If you are using `DjangoObjectType` you can define a custom *get_queryset*.

```
from graphene import relay
from graphene_django.types import DjangoObjectType
from .models import Post

class PostNode(DjangoObjectType):
    class Meta:
        model = Post

    @classmethod
    def get_queryset(cls, queryset, info):
        if info.context.user.is_anonymous:
            return queryset.filter(published=True)
        return queryset
```

1.11.5 Filtering ID-based Node Access

In order to add authorization to id-based node access, we need to add a method to your `DjangoObjectType`.

```
from graphene_django.types import DjangoObjectType
from .models import Post

class PostNode(DjangoObjectType):
    class Meta:
        model = Post
        fields = ('title', 'content')
        interfaces = (relay.Node, )

    @classmethod
    def get_node(cls, info, id):
        try:
            post = cls._meta.model.objects.get(id=id)
        except cls._meta.model.DoesNotExist:
            return None

        if post.published or info.context.user == post.owner:
            return post
        return None
```

1.11.6 Adding Login Required

To restrict users from accessing the GraphQL API page the standard Django LoginRequiredMixin can be used to create your own standard Django Class Based View, which includes the `LoginRequiredMixin` and subclasses the `GraphQLView`.:

```
# views.py

from django.contrib.auth.mixins import LoginRequiredMixin
from graphene_django.views import GraphQLView


class PrivateGraphQLView(LoginRequiredMixin, GraphQLView):
    pass
```

After this, you can use the new `PrivateGraphQLView` in the project's URL Configuration file `url.py`:

For Django 1.11:

```
urlpatterns = [
  # some other urls
  url(r'^graphql$', PrivateGraphQLView.as_view(graphiql=True, schema=schema)),
]
```

For Django 2.0 and above:

```
urlpatterns = [
  # some other urls
  path('graphql', PrivateGraphQLView.as_view(graphiql=True, schema=schema)),
]
```

1.12 Django Debug Middleware

You can debug your GraphQL queries in a similar way to django-debug-toolbar, but outputting in the results in GraphQL response as fields, instead of the graphical HTML interface.

For that, you will need to add the plugin in your graphene schema.

1.12.1 Installation

For use the Django Debug plugin in Graphene:

- Add `graphene_django.debug.DjangoDebugMiddleware` into `MIDDLEWARE` in the `GRAPHENE` settings.
- Add the `debug` field into the schema root `Query` with the value `graphene.Field(DjangoDebug, name='_debug')`.

```
from graphene_django.debug import DjangoDebug

class Query(graphene.ObjectType):
    # ...
    debug = graphene.Field(DjangoDebug, name='_debug')

schema = graphene.Schema(query=Query)
```

And in your `settings.py`:

```
GRAPHENE = {
    ...
    'MIDDLEWARE': [
        'graphene_django.debug.DjangoDebugMiddleware',
    ]
}
```

1.12.2 Querying

You can query it for outputting all the sql transactions that happened in the GraphQL request, like:

```
{
  # A example that will use the ORM for interact with the DB
  allIngredients {
    edges {
      node {
        id,
        name
      }
    }
  }
  # Here is the debug field that will output the SQL queries
  _debug {
    sql {
      rawSql
    }
  }
}
```

Note that the `_debug` field must be the last field in your query.

1.13 Introspection Schema

Relay Modern uses Babel Relay Plugin which requires you to provide your GraphQL schema data.

Graphene comes with a Django management command to dump your schema data to `schema.json` which is compatible with babel-relay-plugin.

1.13.1 Usage

Include `graphene_django` to `INSTALLED_APPS` in your project settings:

```
INSTALLED_APPS += ('graphene_django')
```

Assuming your Graphene schema is at `tutorial.quickstart.schema`, run the command:

```
./manage.py graphql_schema --schema tutorial.quickstart.schema --out schema.json
```

It dumps your full introspection schema to `schema.json` inside your project root directory. Point `babel-relay-plugin` to this file and you're ready to use Relay with Graphene GraphQL implementation.

The schema file is sorted to create a reproducible canonical representation.

1.13.2 GraphQL SDL Representation

The schema can also be exported as a GraphQL SDL file by changing the file extension :

```
./manage.py graphql_schema --schema tutorial.quickstart.schema --out schema.graphql
```

When exporting the schema as a `.graphql` file the `--indent` option is ignored.

1.13.3 Advanced Usage

The `--indent` option can be used to specify the number of indentation spaces to be used in the output. Defaults to *None* which displays all data on a single line.

The `--watch` option can be used to run `./manage.py graphql_schema` in watch mode, where it will automatically output a new schema every time there are file changes in your project

To simplify the command to `./manage.py graphql_schema`, you can specify the parameters in your settings.py:

```
GRAPHENE = {
    'SCHEMA': 'tutorial.quickstart.schema',
    'SCHEMA_OUTPUT': 'data/schema.json',  # defaults to schema.json,
    'SCHEMA_INDENT': 2,  # Defaults to None (displays all data on a single line)
}
```

Running `./manage.py graphql_schema` dumps your schema to `<project root>/data/schema.json`.

1.13.4 Help

Run `./manage.py graphql_schema -h` for command usage.

1.14 Testing API calls with django

1.14.1 Using unittest

If you want to unittest your API calls derive your test case from the class *GraphQLTestCase*.

Your endpoint is set through the *GRAPHQL_URL* attribute on *GraphQLTestCase*. The default endpoint is *GRAPHQL_URL = "/graphql/"*.

Usage:

```
import json

from graphene_django.utils.testing import GraphQLTestCase

class MyFancyTestCase(GraphQLTestCase):
    def test_some_query(self):
        response = self.query(
            '''
            query {
                myModel {
                    id
                    name
                }
            }
            ''',
            op_name='myModel'
        )

        content = json.loads(response.content)

        # This validates the status code and if you get errors
        self.assertResponseNoErrors(response)

        # Add some more asserts if you like
        ...

    def test_query_with_variables(self):
        response = self.query(
            '''
            query myModel($id: Int!){
                myModel(id: $id) {
                    id
                    name
                }
            }
            ''',
            op_name='myModel',
            variables={'id': 1}
        )

        content = json.loads(response.content)
```

```
        # This validates the status code and if you get errors
        self.assertResponseNoErrors(response)

        # Add some more asserts if you like
        ...

    def test_some_mutation(self):
        response = self.query(
            '''
            mutation myMutation($input: MyMutationInput!) {
                myMutation(input: $input) {
                    my-model {
                        id
                        name
                    }
                }
            }
            ''',
            op_name='myMutation',
            input_data={'my_field': 'foo', 'other_field': 'bar'}
        )

        # This validates the status code and if you get errors
        self.assertResponseNoErrors(response)

        # Add some more asserts if you like
        ...
```

1.14.2 Using pytest

To use pytest define a simple fixture using the query helper below

```
# Create a fixture using the graphql_query helper and `client` fixture from `pytest-
↪django`.
import json
import pytest
from graphene_django.utils.testing import graphql_query

@pytest.fixture
def client_query(client):
    def func(*args, **kwargs):
        return graphql_query(*args, **kwargs, client=client)

    return func

# Test you query using the client_query fixture
def test_some_query(client_query):
    response = client_query(
        '''
        query {
            myModel {
                id
                name
            }
        }
```

```
        ''',
        op_name='myModel'
    )

    content = json.loads(response.content)
    assert 'errors' not in content
```

1.15 Settings

Graphene-Django can be customised using settings. This page explains each setting and their defaults.

1.15.1 Usage

Add settings to your Django project by creating a Dictonary with name `GRAPHENE` in the project's `settings.py`:

```
GRAPHENE = {
    ...
}
```

1.15.2 SCHEMA

The location of the top-level `Schema` class.

Default: `None`

```
GRAPHENE = {
    'SCHEMA': 'path.to.schema.schema',
}
```

1.15.3 SCHEMA_OUTPUT

The name of the file where the GraphQL schema output will go.

Default: `schema.json`

```
GRAPHENE = {
    'SCHEMA_OUTPUT': 'schema.json',
}
```

1.15.4 SCHEMA_INDENT

The indentation level of the schema output.

Default: `2`

```
GRAPHENE = {
    'SCHEMA_INDENT': 2,
}
```

1.15.5 MIDDLEWARE

A tuple of middleware that will be executed for each GraphQL query.

See the middleware documentation for more information.

Default: `()`

```
GRAPHENE = {
    'MIDDLEWARE': (
        'path.to.my.middleware.class',
    ),
}
```

1.15.6 RELAY_CONNECTION_ENFORCE_FIRST_OR_LAST

Enforces relay queries to have the `first` or `last` argument.

Default: `False`

```
GRAPHENE = {
    'RELAY_CONNECTION_ENFORCE_FIRST_OR_LAST': False,
}
```

1.15.7 RELAY_CONNECTION_MAX_LIMIT

The maximum size of objects that can be requested through a relay connection.

Default: `100`

```
GRAPHENE = {
    'RELAY_CONNECTION_MAX_LIMIT': 100,
}
```

1.15.8 CAMELCASE_ERRORS

When set to `True` field names in the `errors` object will be camel case. By default they will be snake case.

Default: `False`

```
GRAPHENE = {
   'CAMELCASE_ERRORS': False,
}

# result = schema.execute(...)
print(result.errors)
# [
#     {
#         'field': 'test_field',
#         'messages': ['This field is required.'],
#     }
# ]
```

```
GRAPHENE = {
    'CAMELCASE_ERRORS': True,
}

# result = schema.execute(...)
print(result.errors)
# [
#     {
#         'field': 'testField',
#         'messages': ['This field is required.'],
#     }
# ]
```

1.15.9 DJANGO_CHOICE_FIELD_ENUM_V3_NAMING

Set to `True` to use the new naming format for the auto generated Enum types from Django choice fields. The new format looks like this: `{app_label}{object_name}{field_name}Choices`

Default: `False`

1.15.10 DJANGO_CHOICE_FIELD_ENUM_CUSTOM_NAME

Define the path of a function that takes the Django choice field and returns a string to completely customise the naming for the Enum type.

If set to a function then the `DJANGO_CHOICE_FIELD_ENUM_V3_NAMING` setting is ignored.

Default: `None`

```
# myapp.utils
def enum_naming(field):
    if isinstance(field.model, User):
        return f"CustomUserEnum{field.name.title()}"
    return f"CustomEnum{field.name.title()}"

GRAPHENE = {
    'DJANGO_CHOICE_FIELD_ENUM_CUSTOM_NAME': "myapp.utils.enum_naming"
}
```

1.15.11 SUBSCRIPTION_PATH

Define an alternative URL path where subscription operations should be routed.

The GraphiQL interface will use this setting to intelligently route subscription operations. This is useful if you have more advanced infrastructure requirements that prevent websockets from being handled at the same path (e.g., a WSGI server listening at `/graphql` and an ASGI server listening at `/ws/graphql`).

Default: `None`

```
GRAPHENE = {
    'SUBSCRIPTION_PATH': "/ws/graphql"
}
```

1.15.12 GRAPHIQL_HEADER_EDITOR_ENABLED

GraphiQL starting from version 1.0.0 allows setting custom headers in similar fashion to query variables.

Set to `False` if you want to disable GraphiQL headers editor tab for some reason.

This setting is passed to `headerEditorEnabled` GraphiQL options, for details refer to GraphiQLDocs.

Default: `True`

```
GRAPHENE = {
    'GRAPHIQL_HEADER_EDITOR_ENABLED': True,
}
```