Graphene Documentation

Release 1.0.dev

Syrus Akbary

Sep 05, 2018

Contents

Contents:

CHAPTER 1

Introduction tutorial - Graphene and Django

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

Our primary focus here is to give a good understanding of how to connect models from Django ORM to graphene object types.

A good idea is to check the graphene documentation first.

1.1 Set up the Django project

You can find the entire project in `examples/cookbook-plain`.

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
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...

1.1.1 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:

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

Don't forget to create & run migrations:

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

1.1.2 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`).

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 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.

This means, for each of our models, we are going to create a type, subclassing `DjangoObjectType`

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

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

```
# cookbook/ingredients/schema.py
import graphene

from graphene_django.types import DjangoObjectType

from cookbook.ingredients.models import Category, Ingredient


class CategoryType(DjangoObjectType):
    class Meta:
        model = Category


class IngredientType(DjangoObjectType):
    class Meta:
        model = Ingredient


class Query(object):
    all_categories = graphene.List(CategoryType)
    all_ingredients = graphene.List(IngredientType)

    def resolve_all_categories(self, info, **kwargs):
        return Category.objects.all()

    def resolve_all_ingredients(self, info, **kwargs):
        # We can easily optimize query count in the resolve method
        return Ingredient.objects.select_related('category').all()
```

Note that the above `Query` class is a mixin, inheriting from `object`. This is because we will now create a project-level query class which will combine all our app-level mixins.

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

```
import graphene

import cookbook.ingredients.schema


class Query(cookbook.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 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.

1.3.1 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`:

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

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.

1.3.2 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`.

```
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)),
]
```

1.3.3 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 1.9, 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"
```

(continues on next page)

(continued from previous page)

```
      },
      {
        "id": "3",
        "name": "Beef"
      },
      {
        "id": "4",
        "name": "Chicken"
      }
    ]
  }
}
```

You can experiment with `allCategories` too.

Something to have in mind is the auto camelcasing that is happening.

1.4 Getting relations

Right now, with this simple setup in place, we can query for relations too. This is where graphql becomes really powerful!

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

We can do that with the following query:

```
query {
  allCategories {
    id
    name
    ingredients {
      id
      name
    }
  }
}
```

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

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

(continues on next page)

(continued from previous page)

```
      },
      {
        "id": "2",
        "name": "Meat",
        "ingredients": [
          {
            "id": "3",
            "name": "Beef"
          },
          {
            "id": "4",
            "name": "Chicken"
          }
        ]
      }
    ]
  }
}
```

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

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

1.5 Getting single objects

So far, we have been able to fetch list of objects and follow relation. But what about single objects?

We can update our schema to support that, by adding new query for `ingredient` and `category` and adding arguments, so we can query for specific objects.

```
import graphene

from graphene_django.types import DjangoObjectType

from cookbook.ingredients.models import Category, Ingredient


class CategoryType(DjangoObjectType):
    class Meta:
        model = Category


class IngredientType(DjangoObjectType):
    class Meta:
        model = Ingredient
```

(continues on next page)

(continued from previous page)

```
class Query(object):
    category = graphene.Field(CategoryType,
                              id=graphene.Int(),
                              name=graphene.String())
    all_categories = graphene.List(CategoryType)


    ingredient = graphene.Field(IngredientType,
                                id=graphene.Int(),
                                name=graphene.String())
    all_ingredients = graphene.List(IngredientType)

    def resolve_all_categories(self, info, **kwargs):
        return Category.objects.all()

    def resolve_all_ingredients(self, info, **kwargs):
        return Ingredient.objects.all()

    def resolve_category(self, info, **kwargs):
        id = kwargs.get('id')
        name = kwargs.get('name')

        if id is not None:
            return Category.objects.get(pk=id)

        if name is not None:
            return Category.objects.get(name=name)

        return None

    def resolve_ingredient(self, info, **kwargs):
        id = kwargs.get('id')
        name = kwargs.get('name')

        if id is not None:
            return Ingredient.objects.get(pk=id)

        if name is not None:
            return Ingredient.objects.get(name=name)

        return None
```

Now, with the code in place, we can query for single objects.

For example, lets query `category`:

```
query {
  category(id: 1) {
    name
  }
  anotherCategory: category(name: "Dairy") {
    ingredients {
      id
      name
    }
```

(continues on next page)

(continued from previous page)

```
  }
}
```

This will give us the following results:

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

As an exercise, you can try making some queries to `ingredient`.

Something to keep in mind - since we are using one field several times in our query, we need aliases

1.6 Summary

As you can see, GraphQL is very powerful but there are a lot of repetitions in our example. We can do a lot of improvements by adding layers of abstraction on top of `graphene-django`.

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

CHAPTER 2

Graphene and Django Tutorial using Relay

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

2.1 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...

2.1.1 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
```

2.1.2 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`).

2.2 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(object):
    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
```

(continues on next page)

(continued from previous page)

```
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).

2.3 Testing everything so far

2.3.1 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.

2.3.2 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),
```

(continues on next page)

(continued from previous page)

```
    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)),
]
```

2.3.3 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 1.9, 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 {
```

(continues on next page)

(continued from previous page)

```
    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
      }
    }
  }
}
```

CHAPTER 3

Filtering

Graphene 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 django-filter
pip install django-filter>=2
```

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

3.1 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
      }
    }
  }
}
```

3.2 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` 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, )
```

(continues on next page)

(continued from previous page)

```
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)
```

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)
```

CHAPTER 4

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')
```

4.1 Limiting Field Access

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

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

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

conversely you can use `exclude_fields` meta attribute.

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

(continues on next page)

(continued from previous page)

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

4.2 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)
```

4.3 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)
```

4.4 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
        only_fields = ('title', 'content')
        interfaces = (relay.Node, )

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

4.5 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`.:

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

For Django 1.9 and below:

```
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)),
]
```

CHAPTER 5

Django Debug Middleware

You can debug your GraphQL queries in a similar way to django-debug-toolbar, but outputing 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.

5.1 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',
    ]
}
```

5.2 Querying

You can query it for outputing 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.

CHAPTER 6

Integration with Django Rest Framework

You can re-use your Django Rest Framework serializer with graphene django.

6.1 Mutation

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
```

6.2 Create/Update Operations

By default ModelSerializers accept create and update operations. To customize this use the *model_operations* attribute. The update operation looks up models by the primary key by default. You can customize the look up with the lookup attribute.

```
from graphene_django.rest_framework.mutation import SerializerMutation

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

6.3 Overriding Update Queries

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

```
from graphene_django.rest_framework.mutation import SerializerMutation

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}
```

CHAPTER 7

Integration with Django forms

Graphene-Django comes with mutation classes that will convert the fields on Django forms into inputs on a mutation.
Note: the API is experimental and will likely change in the future.

7.1 FormMutation

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

class MyMutation(FormMutation):
    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.

7.2 ModelFormMutation

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

```
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):
```

(continues on next page)

(continued from previous page)

```
    class Meta:
        model = Pet

class PetMutation(DjangoModelFormMutation):
    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'
```

7.3 Form validation

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

If the form is valid then `form_valid(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.

CHAPTER 8

Introspection Schema

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

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

8.1 Usage

Include `graphene_django` to `INSTALLED_APPS` in you 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.

8.2 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.

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
}
```

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

8.3 Help

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