graphene-django-cud

Release 0.11.0

Jul 11, 2023

User Guide

Graphene-django-cud is an extension of graphene-django, supplying a number of helper classes designed to fast-track creation of create, update and delete mutations.

CHAPTER 1

Installation

Installation is done with pip (or via wrappers such as pipenv or poetry):

```
pip install graphene_django_cud
```

CHAPTER 2

Basic usage

To use, here illustrated by `DjangoCreateMutation`, simply create a new inherting class. Suppose we have the following model and Node.

```
class User(models.Model):
    name = models.CharField(max_length=255)
    address = models.TextField()

class UserNode(DjangoObjectType):
    class Meta:
        model = User
        interfaces = (Node,)
```

Then we can create a create mutation with the following schema

```
class CreateUserMutation(DjangoCreateMutation):
    class Meta:
        model = User

class Mutation(graphene.ObjectType):
    create_user = CreateUserMutation.Field()

class Query(graphene.ObjectType):
    user = graphene.Field(UserNode, id=graphene.String())

    def resolve_user(self, info, id):
        return User.objects.get(pk=id)

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

Note that the `UserNode` has to be registered as a field before the mutation is instantiated. This will be configurable in the future.

The input to the mutation is a single variable `input` which is automatically created with the models fields. An example mutation would then be

```
mutation {
    createUser(input: {name: "John Doe", address: "Downing Street 10"}){
        user{
            id
            name
            address
        }
    }
}
```

CHAPTER 3

Mutations

3.1 DjangoCreateMutation

Mutation class for creating a new instance of the supplied model.

The mutation accepts one argument named *input*. The mutation returns a single field for resolving, which is the camel-case version of the model name.

```
class CreateUserMutation(DjangoCreateMutation):
    class Meta:
        model = User
```

```
mutation {
    createUser(input: {name: "John Doe", address: "161 Lexington Avenue"}){
        user{
            id
            name
            address
        }
    }
}
```

3.2 DjangoUpdateMutation

Mutation class for updating an existing instance of the supplied model.

The mutation accepts two arguments named *id*, and *input*. The mutation returns a single field for resolving, which is the camel-case version of the model name.

The type of the *id* argument is *ID*. However, both regular primary keys and relay global id's are accepted and handled properly.

By default, all *included fields* of the model are marked as required in the input.

```
class UpdateUserMutation(DjangoUpdateMutation):
    class Meta:
        model = User
```

```
mutation {
    updateUser(id: "VXNlck5vZGU6MQ==", input: {name: "John Doe", address: "161␣
↪Lexington Avenue"}){
        user{
            id
            name
            address
        }
    }
}
```

3.3 DjangoPatchMutation

Mutation class for updating an existing instance of the supplied model.

The mutation accepts two arguments named *id*, and *input*. The mutation returns a single field for resolving, which is the camel-case version of the model name.

The type of the *id* argument is *ID*. However, both regular primary keys and relay global id's are accepted and handled properly.

All fields of the model are marked as **not required**.

```
class PatchUserMutation(DjangoPatchMutation):
    class Meta:
        model = User
```

```
mutation {
    patchUser(id: "VXNlck5vZGU6MQ==", input: {name: "John Doe"}){
        user{
            id
            name
            address
        }
    }
}
```

3.4 DjangoDeleteMutation

Mutation class for deleting a single instance of the supplied model.

The mutation accepts one argument named *id*. The type of the *id* argument is *ID*. However, both regular primary keys and relay global id's are accepted and handled properly.

The mutation returns two fields for resolving:

- `found`: True if the instance was found and deleted.
- `deletedId`: The id (primary key) of the deleted instance.

```
class DeleteUserMutation(DjangoDeleteMutation):
    class Meta:
        model = User
```

```
mutation {
    deleteUser(id: "VXNlck5vZGU6MTMzNw=="){
        found
        deletedId
    }
}
```

3.5 DjangoBatchCreateMutation

Mutation class for creating multiple new instances of the supplied model.

The mutation accepts one argument named *input*, which is an array-version of the typical create-input. The mutation returns a single field for resolving, which is the camel-case version of the model name.

```
class BatchCreateUserMutation(DjangoBatchCreateMutation):
    class Meta:
        model = User
```

```
mutation {
    batchCreateUser(input: {name: "John Doe", address: "161 Lexington Avenue"}){
        user{
            id
            name
            address
        }
    }
}
```

3.6 DjangoBatchUpdateMutation

Mutation class for update multiple instances of the supplied model.

The mutation accepts one argument named *input*, which is an array-version of the typical update-input, with the addition that all object IDs are inside the objects. The mutation returns a single field for resolving, which is the camel-case version of the model name.

```
class BatchUpdateUserMutation(DjangoBatchUpdateMutation):
    class Meta:
        model = User
```

```
mutation {
    batchUpdateUser(input: [{
        id: "VXNlck5vZGU6MTMzNw==",
        name: "John Doe",
        address: "161 Lexington Avenue"
    }]){
        user{
```

(continues on next page)

(continued from previous page)

```
            id
            name
            address
        }
    }
}
```

3.7 DjangoBatchPatchMutation

Mutation class for patching multiple instances of the supplied model.

The mutation accepts one argument named *input*, which is an array-version of the typical update-input, with the addition that all object IDs are inside the objects. The mutation returns a single field for resolving, which is the camel-case version of the model name.

```
class BatchPatchUserMutation(DjangoBatchPatchMutation):
    class Meta:
        model = User
```

```
mutation {
    batchPatchUser(input: [{
        id: "VXNlck5vZGU6MTMzNw==",
        address: "161 Lexington Avenue"
    }]){
        user{
            id
            name
            address
        }
    }
}
```

3.8 DjangoFilterUpdateMutation

Mutation class for updating multiple instances of the supplied model. The filtering used to decide which instances to update, is defined in the meta-attribute *filter_fields*.

The mutation accepts two arguments named *filter* and *data*. The shape of *filter* is based on the contents of *filter_fields*. The fields, and their input, is passed directly to an *Model.objects.filter*-call.

The shape of *data* is similar to a DjangoUpdateMutation *input* field, although all fields are optional by default.

The mutation returns two fields for resolving:

- `updatedCount`: The number of updated objects.
- `updatedObjects`: The updated objects.

```
class FilterUpdateUserMutation(DjangoFilterUpdateMutation):
    class Meta:
        model = User
        filter_fields = (
            "name",
```

(continues on next page)

(continued from previous page)

```
            "house__address",
            "house__owner__name__in"
        )
```

```
mutation {
    filterUpdateUsers(
        filter: {
            "name": "John Doe",
            "house_Owner_Name_In": ["Michael Bloomberg", "Steve Jobs"]
        },
        data: {
            "name": "New name"
        }
    ){
        updatedObjects{
            id
            name
        }
    }
}
```

3.9 DjangoFilterDeleteMutation

Mutation class for deleting multiple instances of the supplied model. The filtering used to decide which instances to delete, is defined in the meta-attribute *filter_fields*.

The mutation accepts one argument named *input*. The shape of *input* is based on the contents of *filter_fields*. The fields, and their input, is passed directly to an *Model.objects.filter*-call.

The mutation returns two fields for resolving:

- `deletionCount`: True if the instance was found and deleted.
- `deletedIds`: The id (primary key) of the deleted instance.

```
class BatchDeleteUserMutation(DjangoBatchDeleteMutation):
    class Meta:
        model = User
        filter_fields = (
            "name",
            "house__address",
            "house__owner__name__in"
        )
```

```
mutation {
    batchDeleteUser(input: {"name": "John Doe", "house_Owner_Name_In": ["Michael␣
↪Bloomberg", "Steve Jobs"]}){
        user{
            id
            name
            address
        }
    }
}
```

3.10 DjangoBatchDeleteMutation

Mutation class for deleting multiple instances of the supplied model.

The mutation accepts one argument named *ids*, which is an array of object IDs.

The mutation returns two fields for resolving:

- `deletionCount`: The number of deleted instances.
- `deletedIds`: The id (primary key) of the deleted instance.
- `missedIds`: The id (primary key) of the instances not found.

```
class BatchDeleteUserMutation(DjangoBatchDeleteMutation):
    class Meta:
        model = User
```

```
mutation {
    batchDeleteUser(ids: [
        "VXNlck5vZGU6MTMzNw=="
    ]){
        user{
            id
            name
            address
        }
    }
}
```

CHAPTER 4

Included and excluded fields

This section is primarily relevant for create, update and patch mutations.

4.1 Excluded fields

When the mutation input types are created, all model fields are iterated over, and added to the input object with the corresponding type. Some fields, such as the `password` field of the standard User model, should in most scenarios be excluded. This can be achieved with the `exclude_fields` attribute:

```
class CreateUserMutation(DjangoCreateMutation):
    class Meta:
        model = User
        exclude_fields = ("password",)
```

4.2 Only fields

In some scenarios, if we have a lot of fields excluded, we might want to supply a list of fields that should be included, and let all others be excluded. This can be achieved with the `only_fields` attribute:

```
class CreateUserMutation(DjangoCreateMutation):
    class Meta:
        model = User
        only_fields = ("first_name","last_name","address",)
```

If both `only_fields` and `exclude_fields` are supplied, first the fields matching `only_fields` are extracted, and then the fields matching `exclude_fields` are removed from this list.

CHAPTER 5

Optional and required fields

This section is primarily relevant for create, update and patch mutations.

5.1 General rules

There are certain rules which decide whether or not a field is marked as required. For patch mutations, all fields are always marked as optional. For update and create mutations, however, the following rules apply:

1. If the field has an *explicit override*, this is used.
2. If the field has a *default*-value, it is marked as optional.
3. If the field is a many-to-many field and has *blank=True*, it is marked as optional.
4. If the field is nullable, it is marked as optional.
5. In all other scenarios, the field is marked as required.

5.2 Explicitly overriding

A field can explicitly be marked as optional or required with the meta-attributes `optional_fields` and `required_fields`:

```
class CreateUserMutation(DjangoCreateMutation):
    class Meta:
        model = User
        required_fields = ("first_name",)
        optional_fields = ("last_name",)
```

CHAPTER 6

Permissions and authentication

6.1 Main attributes

By default, a mutation is accessible by anything and everyone. To add access-control to a mutation, the meta-attributes *permissions* and *login_required* is used.

```
class CreateUserMutation(DjangoCreateMutation):
    class Meta:
        model = User
        login_required = True
        permissions = ("users.add_user",)


class UpdateUserMutation(DjangoUpdateMutation):
    class Meta:
        model = User
        permissions = ("users.change_user", "users.some_custom_perm")
```

Note that having a permissions *typically* (but not necessarily) implies that the user is authenticated. Hence in many cases, simply setting the permissions-array to something is sufficient to guarantee that the user is authenticated.

6.2 The `get_permissions` method

In some scenarios, we might want to grant permission to a mutation conditionally. For this, we can override the `get_permissions` classmethod, which by default simply returns the `permissions`-iterable.

Say for example, we want to grant access to update a user-object if the calling user is the same as the updated user, or if the calling user has the `users.change_user`-permission:

```
class UpdateUserMutation(DjangoUpdateMutation):
    class Meta:
        model = User
```

(continues on next page)

(continued from previous page)

```
        login_required = True
        permissions = ("users.change_user",)

    @classmethod
    def get_permissions(cls, root, info, input, id) -> Iterable[str]:
        # Use the disambiguate_id utility from graphene_django_cud to parse the id
        if int(disambiguate_id(id)) == info.context.user.id:
            # Returning an empty array is essentially the same as granting access␣
↪here.
            return []
        return cls._meta.permissions
```

The `get_permissions` method takes slightly different arguments depending on what mutation is being used. For patch and update mutations, the method is given `(root, info, input, id)`. For create mutations, the method is given `(root, info, input)`.

6.3 Overriding the permissions pipeline

Internally, all mutations call a method called `check_permissions` when checking permissions. The default implementation of this method simply calls the `get_permissions`-method, and checks these permissions against the calling user.

`check_permissions` will by default raise an exception if the calling user does not have the required permissions.

If some other pipeline is desired for checking permissions, you can override the `check_permissions`-method. For instance, we *could* implement the permissions-checking above in the following manner:

```
class UpdateUserMutation(DjangoUpdateMutation):
    class Meta:
        model = User
        login_required = True

    @classmethod
    def check_permissions(cls, root, info, input, id):
        if int(disambiguate_id(id)) == info.context.user.id \
           or info.context.user.has_perm("users.change_user"):
            # Not raising an Exception means the calling user has permission to␣
↪access the mutation
            return

        raise GraphQLError("You do not have permission to access this mutation.")
```

You can also wrap `check_permissions` in decorators, if you so desire.

The `check_permissions` method takes slightly different arguments depending on what mutation is being used. For patch and update mutations, the method is given `(root, info, input, id)`. For create mutations, the method is given `(root, info, input)`.

6.4 Wrapping the mutate method

If none of the above is sufficient, the final frontier is overriding the `mutate`-method of each mutation class. Note that that `check_permissions` takes essentially the same arguments as `mutate`. Hence overriding `mutate` should only be required in very fringe scenarios.

```
class UpdateUserMutation(DjangoUpdateMutation):
    class Meta:
        model = User
        login_required = True

    @classmethod
    def mutate(cls, root, info, input, id):
        if int(disambiguate_id(id)) != info.context.user.id \
           and not info.context.user.has_perm("users.change_user"):
            raise GraphQLError("You do not have permission to access this mutation.")

        return super().mutate(root, info, input, id)
```

CHAPTER 7

Field validation

7.1 Individual fields

Before the mutation is executed, the value of each field is validated. By default, each field passes this validation process. Custom validation can be added per field by adding a `validate_<fieldname>`-method to the mutation class.

```
nordic_names = ["Odin", "Tor", "Balder"]

class CreateUserMutation(DjangoCreateMutation):
    class Meta:
        model = User

    def validate_first_name(root, info, value, input, **kwargs):
        if not value in nordic_names:
            raise ValueError("First name must be nordic")
```

Raise an error if a field does not pass validation.

A field validation function always receives the arguments `(root, info, value, input)`. For some mutations, extra keyword arguments are also supplied:

- *DjangoUpdateMutation* and *DjangoPatchMutation*: `obj`, the retrieved model instance, and `id` the input id.
- *DjangoBatchCreateMutation*: `full_input`, the full input object (i.e. containing all objects to be created).

7.2 Overriding the validation pipeline

Internally, each mutation calls a method named *validate*, which in turn finds the individual field validation methods on the class, and calls these.

You can, however, override this *validate* function, if you need a more complex validation pipeline.

```
class UpdateUserMutation(DjangoUpdateMutation):
    class Meta:
        model = User

    @classmethod
    def validate_first_name(cls, root, info, value, input, **kwargs):
        if not value in nordic_names:
            raise ValueError("First name must be nordic")

    @classmethod
    def validate(cls, root, info, input, obj=None, id=None):
        # Check that the user being updated is active
        if obj and obj.is_active == False:
            raise ValueError("Inactive users cannot be updated")

        super().validate(root, info, input, obj=obj, id=id)
```

The validate method takes the same arguments as the individual validate_field methods, minus the *value* field.

7.3 Known limitations

There is currently no way to explicitly validate nested fields, beyond validating the entire field substructure. I.e. for a deeply nested field named `enemies`, the only way to validate this field and its "sub"-fields, is by having a method `validate_enemies`.

CHAPTER 8

Nested fields

There are four meta fields which allow us to extend the handling of both sides of a foreign key relationship (foreign key extras, many to one extras and one to one extras), as well as many to many relationships.

8.1 Foreign key extras

The `foreign_key_extras` field is a dictionary containing information regarding how to handle a model's foreign keys. Here is an example:

```
class Cat(models.Model):
    owner = models.ForeignKey(User, on_delete=models.CASCADE, related_name="cats")
    name = models.TextField()

class CreateCatMutation(DjangoCreateMutation):
    class Meta:
        model = Cat
        foreign_key_extras = {"owner": {"type": "CreateUserInput"}}
```

By default, the `owner` field is of type `ID!`, i.e. you have to supply the ID of an owner when creating a cat. But suppose you instead for every cat want to create a new user as well. Well that's exactly what this mutation allows for (demands).

Here, the `owner` field will now be of type `CreateUserInput!`, which has to have been created before, typically via a `CreateUserMutation`, which by default will result in the type name `CreateUserInput`. An example call to the mutation is:

```
mutation {
    createCat(input: {owner: {name: "John Doe"}, name: "Kitty"}){
        cat{
            name
            owner {
                id
                name
```

(continues on next page)

(continued from previous page)

```
            }
        }
    }
}
```

If you don't have an existing type for creating a user, e.g. the "CreateUserInput" we used above, you can set the type to "auto", which will create a new type.

8.2 Many to one extras

The `many_to_one_extras` field is a dictionary containing information regarding how to handle many to one relations, i.e. the "other" side of a foreign key. Suppose we have the `Cat` model as above. Looking from the User-side, we could add nested creations of Cat's, by the following mutation

```
class CreateUserMutation(DjangoCreateMutation):
    class Meta:
        model = User
        many_to_one_extras = {
            "cats": {
                "add": {"type": "auto"}
            }
        }
```

This will add an input argument `catsAdd`, which accepts an array of Cat objects. Note that the type value `auto` means that a new type to accept the cat object will be created. This is usually necessary, as the regular `CreateCatInput` requires an owner id, which we do not want to give here, as it is inferred.

Now we could create a user with multiple cats in one go as follows:

```
mutation {
    createUser(input: {
        name: "User",
        catsAdd: [
            {name: "First Kitty"},
            {name: "Second kitty"}
        ]
    }){
        user{
            id
            name
            cats{
                edges{
                    node{
                        id
                    }
                }
            }
        }
    }
}
```

Note that the default many to one relation argument `cats` still accepts a list of inputs. You might want to keep it this way. However, you can override the default by adding an entry with the key "exact":

```
class CreateUserMutation(DjangoCreateMutation):
    class Meta:
        model = User
        many_to_one_extras = {
            "cats": {
                "exact": {"type": "auto"}
            }
        }
```

Note that we can add a new key with the type "ID", to still allow for Cat objects to be added by id.

```
class CreateUserMutation(DjangoCreateMutation):
    class Meta:
        model = User
        many_to_one_extras = {
            "cats": {
                "exact": {"type": "auto"},
                "by_id": {"type": "ID"}
            }
        }
```

```
mutation {
    createUser(input: {
        name: "User",
        cats: [
            {name: "First Kitty"},
            {name: "Second kitty"}
        ],
        catsById: ["Q2F0Tm9kZTox"]
    }){
        user{
            ...UserInfo
        }
    }
}
```

8.3 Many to many extras

The `many_to_many_extras` field is a dictionary containing information regarding how to handle many to many relations. Suppose we have the `Cat` model as above, and a `Dog` model like:

```
class Dog(models.Model):
    owner = models.ForeignKey(User, null=True, on_delete=models.SET_NULL)
    name = models.TextField()

    enemies = models.ManyToManyField(Cat, blank=True, related_name='enemies')

    def is_stray():
        return self.owner is None


class DogNode(DjangoObjectType):
    class Meta:
        model = Dog
```

We now have a many to many relationship, which by default will be modelled by default using an `[ID]` argument. However, this can be customized fairly similar to many to one extras:

```
class CreateDogMutation(DjangoCreateMutation):
    class Meta:
        model = Dog
        many_to_many_extras = {
            'enemies': {
                'add': {"type": "CreateCatInput"}
            }
        }
```

This will, similar to before, add an `enemiesAdd` argument:

```
mutation {
    createDog(input: {
        name: "Buster",
        enemies: ["Q2F0Tm9kZTox"],
        enemiesAdd: [{owner: "VXNlck5vZGU6MQ==", name: "John's cat"]
    }){
        dog{
            ...DogInfo
        }
    }
}
```

This will create a dog with two enemies, one that already exists, and a new one, which has the owner `VXNlck5vZGU6MQ==` (some existing user). Note that if `CreateCatInput` expects us to create a new user, we would have to do that here.

We can also add an extra field here for removing entities from a many to many relationship:

```
class UpdateDogMutation(DjangoUpdateMutation):
    class Meta:
        model = Dog
        many_to_many_extras = {
            "enemies": {
                "add": {"type": "CreateCatInput"},
                "remove": {"type": "ID"},
                # A similar form would be "remove": true
            }
        }
```

Note that this *has* to have the type "ID". Also note that this has no effect on `DjangoCreateMutation` mutations. We could then perform

```
mutation {
    updateDog(id: "RG9nTm9kZTox", input: {
        name: "Buster 2",
        enemiesRemove: ["Q2F0Tm9kZTox"],
        enemiesAdd: [{owner: "VXNlck5vZGU6MQ==", name: "John's cat"]
    }){
        dog{
            ...DogInfo
        }
    }
}
```

This would remove "Q2F0Tm9kZTox" as an enemy, in addition to creating a new one as before.

We can alter the behaviour of the default argument (e.g. `enemies`), by adding the "exact":

```
class UpdateDogMutation(DjangoUpdateMutation):
    class Meta:
        model = Dog
        many_to_many_extras = {
            "enemies": {
                "exact": {"type": "CreateCatInput"},
                "remove": {"type": "ID"},
                # A similar form would be "remove": true
            }
        }
```

```
mutation {
    updateDog(id: "RG9nTm9kZTox", input: {
        name: "Buster 2",
        enemies: [{owner: "VXNlck5vZGU6MQ==", name: "John's cat"]
    }){
        dog{
            ...DogInfo
        }
    }
}
```

This will have the rather odd behavior that all enemies are reset, and only the new ones created will be added to the relationship. In other words it exists as a sort of `purge and create` functionality. When used in a `DjangoCreateMutation` it will simply function as an initial populator of the relationship.

If you don't have an existing type for creating a user, e.g. the "CreateCatInput" we used above, you can set the type to "auto", which will create a new type.

8.4 One to one extras

The `one_to_one_extras` field is a dictionary containing information regarding how to handle a model's One-ToOne fields. Here is an example:

```
class CreateDogMutation(DjangoCreateMutation):
    class Meta:
        model = Dog
        one_to_one_extras = {"registration": {"type": "auto"}}
```

By default, the registration field is a type `ID!`, but using `auto`, this will make a new type to accept create a `registration` object, called `CreateDogCreateRegistrationInput`.

8.5 Other aliases

In both the many to many and many to one extras cases, the naming of the extra fields are not arbitrary. However, they can be customized. Suppose you want your field to be named `enemiesKill`, which should remove from a many to many relationship. Then initially, we might write:

```
class UpdateDogMutation(DjangoUpdateMutation):
    class Meta:
        model = Dog
```

(continues on next page)

(continued from previous page)

```
        many_to_many_extras = {
            "enemies": {
                "exact": {"type": "CreateCatInput"},
                "kill": {"type": "ID"},
            }
        }
```

Unfortunately, this will not work, as graphene-django-cud does not know what operation `kill` translates to. Should we add or remove (or set) the entities? Fortunately, we can explicitly tell which operation to use, by supplying the "operation" key:

```
class UpdateDogMutation(DjangoUpdateMutation):
    class Meta:
        model = Dog
        many_to_many_extras = {
            "enemies": {
                "exact": {"type": "CreateCatInput"},
                "kill": {"type": "ID", "operation": "remove"},
            }
        }
```

Legal values are "add", "remove", and "update" (and some aliases of these).

The argument names can also be customized:

```
class UpdateDogMutation(DjangoUpdateMutation):
    class Meta:
        model = Dog
        many_to_many_extras = {
            "enemies": {
                "exact": {"type": "CreateCatInput"},
                "kill": {"type": "ID", "operation": "remove", "name": "kill_enemies"},
            }
        }
```

The name of the argument will be `killEnemies` instead of the default `enemiesKill`. The name will be translated from snake_case to camelCase as per usual.

8.6 Excluding fields

By default, all fields are included in the input type. However, you can exclude fields by using the `exclude_fields` attribute:

```
class CreateDogMutation(DjangoCreateMutation):
    class Meta:
        model = Dog
        many_to_many_extras = {
            "enemies": {
                "exact": {
                    "type": "CreateCatInput",
                    "exclude_fields": ("name",),
                },
            }
        }
```

This will exclude the `name` field from the input type.

8.7 Deep nested arguments

Note that deeply nested arguments are added by default when using existing types. Hence, for the mutation

```
class CreateDogMutation(DjangoCreateMutation):
    class Meta:
        model = Dog
        many_to_many_extras = {
            "enemies": {
                "exact": {"type": "CreateCatInput"},
            }
        }
```

Where `CreateCatInput` is the type generated for

```
class CreateCatMutation(DjangoCreateMutation):
    class Meta:
        model = Cat
        many_to_many_extras = {
            "targets": {"exact": {"type": "CreateMouseInput"}},
        }
        foreign_key_extras = {"owner": {"type": "CreateUserInput"}}
```

Where we assume we have now also created a new model `Mouse` with a standard `CreateMouseMutation` mutation. We could then execute the following mutation:

```
mutation {
    createDog(input: {
        owner: null,
        name: "Spark",
        enemies: [
            {
                name: "Kitty",
                owner: {name: "John doe"},
                targets: [
                    {name: "Mickey mouse"}
                ]
            },
            {
                name: "Kitty",
                owner: {name: "Ola Nordmann"}
            }
        ]
   }){
        ...DogInfo
   }
}
```

This creates a new (stray) dog, two new cats with one new owner each and one new mouse. The new cats and the new dog are automatically set as enemies, and the mouse is automatically set as a target of the first cat.

For `auto` fields, we can create nested behaviour explicitly:

```
class CreateUserMutation(DjangoCreateMutation):
    class Meta:
        model = User
        many_to_one_extras = {
            "cats": {
                "exact": {
                    "type": "auto",
                    "many_to_many_extras": {
                        "enemies": {
                            "exact": {
                               "type": "CreateDogInput"
                            }
                        }
                    }
                }
            }
        }
```

There is no limit to how deep this recursion may be.

CHAPTER 9

Custom field value handling

9.1 Handlers

In some scenarios, field values have to be handled or transformed in a custom manner before it is saved. For this we can use custom field handlers. To create a custom field handler, add a method to the mutation class named *handle_<fieldname>*.

Suppose we have a user object with a gpa-score field, which we don't bother to validate, but want to clamp between 1.0 and 4.0.

```
class UpdateUserMutation(DjangoUpdateMutation):
    class Meta:
        model = User

    @classmethod
    def handle_gpa(cls, value, name, info) -> int:
        return max(1.0, min(4.0, value))
```

The returned value from a handle-method will be the one used when updating/creating an instance of the model.

Notably, this method will override a few specific internal mechanisms:

- By default, foreign keys fields will have "_id" attached as a suffix to the field name before saving the raw id. Also global relay ids and regular ids are disambiguated.
- Many to many fields which accept IDs are disambiguated in a similar manner.

This will not happen if you add handle-functions for such fields, and hence you are expected to translate the values into values Django understands internally.

NB: The method signature of handle-fields are due to change before version 1.0.0. The new signature will most likely be `(root, info, value, input)`, with `obj`, `id` and `full_input` **as potential extra kwargs.**

9.2 Known limitations

There is currently no way to separately handle nested fields, beyond handling the entire field substructure. I.e. for a deeply nested field named `enemies`, the only way to handle this field and its "sub"-fields, is by having a method `handle_enemies`.

Do note however, that if models have clashing field names, the handle-method will be called for both these fields.

This is something being actively worked on resolving.

CHAPTER 10

Auto context fields

The create, update and patch mutations contains a meta-field `auto_context_fields`. It allows us to automatically assign field values depending on values in the context (i.e. the current `HttpRequest`). Most typically, this will be used to automatically assign the the current user to some field.

Suppose for instance you have the following model:

```
class ForumThread(models.Model):
    created_by = models.ForeignKey(User, on_delete=models.CASCADE)

    # More fields
```

We can then automatically assign the created_by field to the calling user by creating a mutation:

```
class CreateForumThreadMutation(DjangoCreateMutation):
    class Meta:
        auto_context_fields = {
            'created_by': 'user'
        }
```

Presupposing, of course, that the `user` field of the `info.context` (HttpRequest) field is set. This works with any context field. Also note that auto context fields are automatically set as `required=False`, to please Graphene. Finally note that if we add an explicit value to the `createdBy` field when calling the mutation, this value will override the auto context field.

CHAPTER 11

Other hooks

These hooks are class methods of a mutation, which can be overridden with custom behavior.

11.1 before_mutate

Mutation	Arguments	Note
create	cls, root, info, input	1
patch/update	cls, root, info, input, id	1
delete	cls, root, info, id	
batch_create	cls, root, info, input	1
batch_patch/batch_update	cls, root, info, input	1
batch_delete/filter_delete	cls, root, info, input	1

1: The hook can modify and return the `input` object. Returning `None` will cause the mutation to use the original `input`.

11.2 before_save

Mutation	Arguments	Note
create	cls, root, info, input, obj	1
patch/update	cls, root, info, input, id, obj	1
delete	cls, root, info, id, obj	1
batch_create	cls, root, info, input, created_objects	2
batch_patch/batch_update	cls, root, info, input, updated_objects	2
batch_delete	cls, root, info, ids, qs_to_delete	3
filter_delete	cls, root, info, filter_qs	3

1: You can optionally modify and return the ORM object `obj`.
2: You can optionally modify and return the ORM objects in `created_objects` or `updated_objects`.
3: You can optionally modify and return the querysets.

11.3 after_mutate

Mutation	Arguments	Note
create	cls, root, info, input, obj, return_data	1
patch/update	cls, root, info, id, input, obj, return_data	1
delete	cls, root, info, deleted_id, found	
batch_create	cls, root, info, input, created_objs, return_data	1
batch_patch/batch_update	cls, root, info, input, updated_objs, return_data	1
batch_delete/filter_delete	cls, root, info, input, deletion_count, ids	

1: You can modify and return the `return_data` argument.

CHAPTER 12

Field, argument and type naming

There are three different names that have to be specified for each mutation:

- The name of the mutation.
- The name of the input argument(s).
- The name of the input argument type.
- The name of the field that can be resolved.

The first one is always set by you, and the second one is always `input` or `id` (or both).

The two others can be customized by the following meta parameters:

- `type_name`
- `return_field_name`

```
class UpdateUserMutation(DjangoUpdateMutation):
    class Meta:
        model = User
        type_name = "ChangeUserInput"  # Default here would be UpdateUserInput
        return_field_name = "updatedUser"  # Default here would be user


class Mutation(graphene.ObjectType):
    update_user = UpdateUserMutation.Field()
```

```
mutation UpdateUserMutation($input: ChangeUserInput){
    updateUser(input: $input){
        updatedUser{

        }
    }
}
```

Given the existence of GraphQL aliasing, the utility of the latter is questionable.

CHAPTER 13

Overriding field types

This section is primarily relevant for create, update and patch mutations.

By default, graphene-django-cud iterates through all the fields of a model, and converts each field to a corresponding graphene type. This converted type is added to the mutation input argument.

The conversions are typically what you would expect, e.g. `models.CharField` is converted to `graphene.String`.

It is possible to override this conversion, by explicitly providing a **field_types** argument. By default, the field will be coerced when added to the Django model instance. If the desired result is either something more complex than a simple coercion, or the overriding type cannot be coerced into the corresponding Django model field; then you must implement a *custom handler*.

```
class Dog(models.Model):
    owner = models.ForeignKey(User, on_delete=models.CASCADE, related_name='dogs')
    name = models.TextField()
    tag = models.CharField(max_length=16, default="Dog-1", help_text="Non-unique␣
↪identifier for the dog, on the form 'Dog-%d'")


class CreateDogMutation(DjangoCreateMutation):
    class Meta:
        model = Dog
        field_types = {
            "tag": graphene.Int(required=False)
        }

    @classmethod
    def handle_tag(cls, value, *args, **kwargs):
        return "Dog-" + str(value)
```

CHAPTER 14

Custom fields

It is possible to add custom input fields to the following mutations:

- DjangoCreateMutation
- DjangoPatchMutation
- DjangoUpdateMutation
- DjangoBatchCreateMutation
- DjangoBatchPatchMutation
- DjangoBatchUpdateMutation

The custom fields will be added to the top-level *input* input data structure. While the fields will not be used directly in any creation/updating process by the library itself, they can be accessed in all *handle*- and *hook*-methods.

```
class Dog(models.Model):
    name = models.TextField()
    bark_count = models.IntegerField(default=0)

class UpdateDogMutation(DjangoUpdateMutation):
    class Meta:
        model = Dog
        custom_fields = {
            "bark": graphene.Boolean()
        }

    @classmethod
    def before_save(cls, root, info, input, id, obj: Dog):
        if input.get("bark"):
            obj.bark_count += 1
        return obj
```

Running the below mutation will increase the bark count by one:

CHAPTER 15

Reusing types

TODO

CHAPTER 16

Known limitations and quirks

One could wish for an API where you could specify both IDs and objects in a single array for many to many and many to one relations. However, due to GraphQLs strict type system, this is not currently possible — in particular due to the fact that scalars and object types cannot simultaneously be part of a union.

Some workarounds could be implemented for this, but we deem this more dirty than useful.

CHAPTER 17

Lifecycle of a mutation

CHAPTER 18

Models documentation

Documentation for all models.

18.1 DjangoCreateMutation

Will create a new mutation which will create a *new* object of the supplied model.

Mutation input arguments:

Argument	Type
input	Object!

Meta fields:

```
mutation {
    createUser(input: {name: "John Doe", address: "Downing Street 10"}){
        user{
            id
            name
            address
        }
    }
}
```

18.2 DjangoUpdateMutation

Will update an existing instance of a model. The UpdateMutation (in contrast to the PatchMutation) requires all fields to be supplied by default.

Mutation input arguments:

Argument	Type
id	ID!
input	Object!

All meta arguments:

Argument	type	Default	Description
model	Model	None	The model. **Required**.
only_fields	Iterable	None	If supplied, only these fields will be added as input variables for the model
exclude_fields	Iterable	None	If supplied, these fields will be excluded as input variables for the model.
return_field_name	String	None	The name of the return field within the mutation. The default is the camelCased name of the model
permissions	Tuple	None	The permissions required to access the mutation
login_required	Boolean	None	If true, the calling user has to be authenticated
auto_context_fields	Dict	None	A mapping of context values into model fields. See below
optional_fields	Tuple	()	A list of fields which explicitly should have `required=False`
required_fields	Tuple	None	A list of fields which explicitly should have `required=True`
custom_fields	Tuple	None	A list of custom graphene fields which will be added to the model input type.
type_name	String	None	If supplied, the input variable in the mutation will have its typename set to this string. This is useful when creating multiple mutations of the same type for a single model.
many_to_many_extras	Dict	{}	A dict with extra information regarding many-to-many fields. See below.
many_to_one_extras	Dict	{}	A dict with extra information regarding many-to-one relations. See below.
foreign_key_extras	Dict	{}	A dict with extra information regarding foreign key extras.
one_to_one_extras	Dict	{}	A dict with extra information regarding one to one extras.
use_select_for_update	Boolean	True	If true, the queryset will be altered with `select_for_update`, locking the database rows in question. Used to ensure data integrity on updates.

```
mutation {
    updateUser(id: "VXNlck5vZGU6MQ==", input: {
        name: "John Doe",
        address: "Downing Street 10"
    }){
        user{
            id
            name
            address
        }
    }
}
```

18.3 DjangoPatchMutation

Will update an existing instance of a model. The PatchMutation (in contrast to the UpdateMutation) does not require all fields to be supplied. I.e. all are fields are optional.

Mutation input arguments:

Argument	Type
id	ID!
input	Object!

All meta arguments:

Argument	type	Default	Description
model	Model	None	The model. **Required**.
only_fields	Iterable	None	If supplied, only these fields will be added as input variables for the model
exclude_fields	Iterable	None	If supplied, these fields will be excluded as input variables for the model.
return_field_name	String	None	The name of the return field within the mutation. The default is the camelCased name of the model
permissions	Tuple	None	The permissions required to access the mutation
login_required	Boolean	None	If true, the calling user has to be authenticated
auto_context_fields	Dict	None	A mapping of context values into model fields. See below
optional_fields	Tuple	()	A list of fields which explicitly should have `required=False`
required_fields	Tuple	None	A list of fields which explicitly should have `required=True`
custom_fields	Tuple	None	A list of custom graphene fields which will be added to the model input type.
type_name	String	None	If supplied, the input variable in the mutation will have its typename set to this string. This is useful when creating multiple mutations of the same type for a single model.
many_to_many_extras	Dict	{}	A dict with extra information regarding many-to-many fields. See below.
many_to_one_extras	Dict	{}	A dict with extra information regarding many-to-one relations. See below.
foreign_key_extras	Dict	{}	A dict with extra information regarding foreign key extras.
one_to_one_extras	Dict	{}	A dict with extra information regarding one to one extras.
use_select_for_update	Boolean	True	If true, the queryset will be altered with `select_for_update`, locking the database rows in question. Used to ensure data integrity on updates.

18.3.1 Example mutation

```
mutation {
    updateUser(id: "VXNlck5vZGU6MQ==", input: {
        name: "John Doe",
    }){
```

(continues on next page)

(continued from previous page)

```
        user{
            id
            name
            address
        }
    }
}
```

18.4 DjangoDeleteMutation

Will delete an existing instance of a model. The returned arguments are:

- `found`: True if the instance was found and deleted.
- `deletedId`: THe id of the deleted instance.

Mutation input arguments:

Argument	Type
id	ID!

All meta arguments:

Argument	type	Default	Description
model	Model	None	The model. **Required**.
permissions	Tuple	None	The permissions required to access the mutation
login_required	Boolean	None	If true, the calling user has to be authenticated

```
mutation {
    deleteUser(id: "VXNlck5vZGU6MQ=="){
        found
        deletedId
    }
}
```

18.5 DjangoBatchCreateMutation

Will create a new mutation which will create multiple *new* objects of the supplied model.

Mutation input arguments:

Argument	Type
input	[Object]!

Meta fields:

Field	Type	Default	Description
model	Model	None	The model. **Required**.
only_fields	Iterable	None	If supplied, only these fields will be added as input variables for the model
exclude_fields	Iterable	None	If supplied, these fields will be excluded as input variables for the model.
return_field_name	String	None	The name of the return field within the mutation. The default is the camelCased name of the model
permissions	Tuple	None	The permissions required to access the mutation
login_required	Boolean	None	If true, the calling user has to be authenticated
auto_context_fields	Dict	None	A mapping of context values into model fields. See below.
optional_fields	Tuple	()	A list of fields which explicitly should have `required=False`
required_fields	Tuple	None	A list of fields which explicitly should have `required=True`
custom_fields	Tuple	None	A list of custom graphene fields which will be added to the model input type.
type_name	String	None	If supplied, the input variable in the mutation will have its typename set to this string. This is useful when creating multiple mutations of the same type for a single model.
use_type_name	String	None	If supplied, no new input type will be created, and instead the registry will be queried for an input type with that name. Note that supplying this value will invalidate many other arguments, as they are only relevant for creating the new input type.
many_to_many_extras	Dict	{}	A dict with extra information regarding many-to-many fields. See below.
many_to_one_extras	Dict	{}	A dict with extra information regarding many-to-one relations. See below.
foreign_key_extras	Dict	{}	A dict with extra information regarding foreign key extras.
one_to_one_extras	Dict	{}	A dict with extra information regarding one to one extras.

```
mutation{
    batchCreateUser(input: [{name: "John Doe", address: "Downing Street 10"}]){
        user{
            id
            name
            address
        }
    }
}
```

18.6 DjangoBatchUpdateMutation

Will create a new mutation which can be used to update multiple objects of the supplied model.

Mutation input arguments:

Argument	Type
input	[Object]!

Meta fields:

Field	Type	Default	Description
model	Model	None	The model. **Required**.
only_fields	Iterable	None	If supplied, only these fields will be added as input variables for the model
exclude_fields	Iterable	None	If supplied, these fields will be excluded as input variables for the model.
return_field_name	String	None	The name of the return field within the mutation. The default is the camelCased name of the model
permissions	Tuple	None	The permissions required to access the mutation
login_required	Boolean	None	If true, the calling user has to be authenticated
auto_context_fields	Dict	None	A mapping of context values into model fields. See below.
optional_fields	Tuple	()	A list of fields which explicitly should have `required=False`
required_fields	Tuple	None	A list of fields which explicitly should have `required=True`
custom_fields	Tuple	None	A list of custom graphene fields which will be added to the model input type.
type_name	String	None	If supplied, the input variable in the mutation will have its typename set to this string. This is useful when creating multiple mutations of the same type for a single model.
use_type_name	String	None	If supplied, no new input type will be created, and instead the registry will be queried for an input type with that name. Note that supplying this value will invalidate many other arguments, as they are only relevant for creating the new input type.
many_to_many_extras	Dict	{}	A dict with extra information regarding many-to-many fields. See below.
many_to_one_extras	Dict	{}	A dict with extra information regarding many-to-one relations. See below.
foreign_key_extras	Dict	{}	A dict with extra information regarding foreign key extras.
one_to_one_extras	Dict	{}	A dict with extra information regarding one to one extras.

```
mutation{
    batchUpdateUser(input: [{
        id: "VXNlck5vZGU6MQ==",
        name: "John Doe",
        address: "Downing Street 10"
    }]){
        user{
            id
            name
            address
        }
    }
}
```

18.7 DjangoBatchPatchMutation

Will create a new mutation which can be used to patch multiple objects of the supplied model.

Mutation input arguments:

Argument	Type
input	[Object]!

Meta fields:

Field	Type	Default	Description
model	Model	None	The model. **Required**.
only_fields	Iterable	None	If supplied, only these fields will be added as input variables for the model
exclude_fields	Iterable	None	If supplied, these fields will be excluded as input variables for the model.
return_field_name	String	None	The name of the return field within the mutation. The default is the camelCased name of the model
permissions	Tuple	None	The permissions required to access the mutation
login_required	Boolean	None	If true, the calling user has to be authenticated
auto_context_fields	Dict	None	A mapping of context values into model fields. See below.
optional_fields	Tuple	()	A list of fields which explicitly should have `required=False`
required_fields	Tuple	None	A list of fields which explicitly should have `required=True`
custom_fields	Tuple	None	A list of custom graphene fields which will be added to the model input type.
type_name	String	None	If supplied, the input variable in the mutation will have its typename set to this string. This is useful when creating multiple mutations of the same type for a single model.
use_type_name	String	None	If supplied, no new input type will be created, and instead the registry will be queried for an input type with that name. Note that supplying this value will invalidate many other arguments, as they are only relevant for creating the new input type.
many_to_many_extras	Dict	{}	A dict with extra information regarding many-to-many fields. See below.
many_to_one_extras	Dict	{}	A dict with extra information regarding many-to-one relations. See below.
foreign_key_extras	Dict	{}	A dict with extra information regarding foreign key extras.
one_to_one_extras	Dict	{}	A dict with extra information regarding one to one extras.

```
mutation{
    batchPatchUser(input: [{
        id: "VXNlck5vZGU6MQ==",
        name: "John Doe",
    }]){
        user{
            id
            name
            address
        }
    }
}
```

18.8 DjangoBatchDeleteMutation

Will delete multiple instances of a model depending on supplied filters. The returned arguments are:

- `deletionCount`: True if the instance was found and deleted.
- `deletedIds`: The ids of the deleted instances.
- `missedIds`: The ids of the missed instances.

Mutation input arguments:

Argument	Type
ids	[ID]!

All meta arguments:

Argument	type	Default	Description
model	Model	None	The model. **Required**.
permissions	Tuple	None	The permissions required to access the mutation
login_required	Boolean	None	If true, the calling user has to be authenticated
return_field_name	String	None	The name of the return field within the mutation. The default is the camelCased name of the model

```
class BatchDeleteUser(DjangoBatchDeleteMutation):
    class Meta:
        model = User
```

```
mutation {
    batchDeleteUser(ids: ["VXNlck5vZGU6MQ=="]){
        deletedIds
        missedIds
        deletionCount
    }
}
```

18.9 DjangoFilterDeleteMutation

Will delete multiple instances of a model depending on supplied filters. The returned arguments are:

- `deletionCount`: True if the instance was found and deleted.
- `deletedIds`: The ids of the deleted instances.

Mutation input arguments:

Argument	Type
input	Object!

All meta arguments:

Argument	type	Default	Description
model	Model	None	The model. **Required**.
filter_fields	Tuple	()	A number of filter fields which allow us to restrict the instances to be deleted.
permissions	Tuple	None	The permissions required to access the mutation
login_required	Boolean	None	If true, the calling user has to be authenticated

If there are multiple filters, these will be combined with **and**-clauses. For or-clauses, use multiple mutation calls.

```
class FilterDeleteUser(DjangoFilterDeleteMutation):
    class Meta:
        model = User
        filter_fields = ('name', 'house__address',)
```

```
mutation {
    filterDeleteUser(input: {name: 'John'}){
        deletedIds
        deletionCount
    }
}
```

18.10 DjangoFilterUpdateMutation

Will update multiple instances of a model depending on supplied filters. The returned arguments are:

- `updatedCount`: The number of updated instances.
- `updatedObjects`: The ids of the deleted instances.

Mutation input arguments: +————+————+ | Argument | Type | +============+===========+ | filter | Object! | +————+————+ | data | Object! | +————+————+

All meta arguments:

Argument	type	Default	Description
model	Model	None	The model. **Required**.
filter_fields	Tuple	()	A number of filter fields which allow us to restrict the instances to be deleted.
only_fields	Iterable	None	If supplied, only these fields will be added as input variables for the model
exclude_fields	Iterable	None	If supplied, these fields will be excluded as input variables for the model.
return_field_name	String	None	The name of the return field within the mutation. The default is the camelCased name of the model
permissions	Tuple	None	The permissions required to access the mutation
login_required	Boolean	None	If true, the calling user has to be authenticated
auto_context_fields	Dict	None	A mapping of context values into model fields. See below
optional_fields	Tuple	()	A list of fields which explicitly should have `required=False`
required_fields	Tuple	None	A list of fields which explicitly should have `required=True`
type_name	String	None	If supplied, the input variable in the mutation will have its typename set to this string. This is useful when creating multiple mutations of the same type for a single model.

If there are multiple filters, these will be combined with **and**-clauses. For or-clauses, use multiple mutation calls.

```
class FilterUpdateUserMutation(DjangoFilterDeleteMutation):
    class Meta:
        model = User
        filter_fields = ('name',)
```

```
mutation {
    filterUpdateUser(filter: {name: 'John'}, data: {name: 'Ola'}){
        updateObjects{
            id
            name
        }
    }
}
```

CHAPTER 19

Conversion utilities

CHAPTER 20

Custom types