django-gm2m Documentation

Release 1.1

Thomas Khyn

Oct 05, 2020

Contents

Django generic many-to-many field implementation.

This django application exposes a `GM2MField` that combines the features of the standard Django `ManyToManyField` and `GenericForeighKey` and that can be used exactly the same way.

It has been tested with Django 2.2.* and 3.0.* and their matching Python versions (3.5 to 3.8).

If you like django-gm2m and find it useful, you may want to thank me and encourage future development by sending a few mBTC / mBCH / mBSV at this address: `1EwENyR8RV6tMc1hsLTkPURtn5wJgaBfG9`.

Documentation contents:

CHAPTER 1

Quick start

1.1 Installation

As straightforward as it can be, using `pip`:

```
pip install django-gm2m
```

You then need to make sure that `django.contrib.contenttypes` appears somewhere in your `INSTALLED_APPS` setting, and add `gm2m` to it:

```
INSTALLED_APPS = [
    ...
    'django.contrib.contenttypes',
    ...
    'gm2m',
]
```

1.2 First steps

You can use the exposed `GM2MField` exactly the same way as a `ManyToManyField`.

Suppose you have some models describing videos types:

```
>>> from django.db import models
>>>
>>> class Video(models.Model):
>>>     title = models.CharField(max_length=255)
>>>
>>> class Movie(Video):
>>>     pass
>>>
```

(continues on next page)

(continued from previous page)

```
>>> class Documentary(Video):
>>>     pass
```

Now, if you want to have a field for the preferred videos of a User, you simply need to add a default `GM2MField` to the `User` model:

```
>>> from gm2m import GM2MField
>>>
>>> class User(models.Model):
>>>     name = models.CharField(max_length=255)
>>>     preferred_videos = GM2MField()
```

Now you can add videos to the `preferred_videos` set:

```
>>> me = User.objects.create(name='Me')
>>> v_for_vendetta = Movie.objects.create(title='V for Vendetta')
>>>
>>> me.preferred_videos.add(v_for_vendetta)
```

or:

```
>>> me.preferred_videos = [v_for_vendetta]
```

You can obviously mix instances from different models:

```
>>> citizenfour = Documentary.objects.create(title='Citizenfour')
>>> me.preferred_videos = [v_for_vendetta, citizenfour]
```

From a `User` instance, you can fetch all the preferred videos:

```
>>> [v.title for v in me.preferred_videos]
['V for Vendetta', 'Citizenfour']
```

... which you may filter by model using the `Model` or `Model__in` keywords:

```
>>> [v.title for v in me.preferred_videos.filter(Model=Movie)]
['V for Vendetta']
>>> [v.title for v in me.preferred_videos.filter(Model__in=[Documentary])]
['Citizenfour']
```

That's it regarding the basic usage of `django-gm2m`. You'll probably want to have a look at the more advanced *features* it offers.

CHAPTER 2

Features

django-gm2m. . .

- Works like the built-in Django related fields
- Creates one table per relation, like `ManyToManyField`, and not one big table linking anything to anything (django-generic-m2m's default approach)
- Does not require you to modify nor monkey-patch the existing model classes that need to be linked
- Provides automatic *reverse relations* when an instance is added
- Enables related objects *prefetching*
- Allows the use of *Through models*
- Allows you to customize the *deletion* behaviour
- Supports *migrations*

In this page, we'll make use of the models that were described in the *Quick start* section:

```
>>> from django.db import models
>>> from gm2m import GM2MField
>>>
>>> class Video(models.Model):
>>>     pass
>>>
>>> class Movie(Video):
>>>     pass
>>>
>>> class Documentary(Video):
>>>     pass
>>>
>>> class User(models.Model):
>>>     preferred_videos = GM2MField()
>>>
>>>
```

(continues on next page)

(continued from previous page)

```
>>> me = User.objects.create(name='Me')
>>>
>>> v_for_vendetta = Movie.objects.create(title='V for Vendetta')
>>> citizenfour = Documentary.objects.create(title='Citizenfour')
>>>
>>> me.preferred_videos = [v_for_vendetta, citizenfour]
```

2.1 Reverse relations

We've seen how you could access all the `preferred_videos` of a given user. But what if you want to access all the users that have bookmarked a given video? `django-gm2m` provides that out of the box, with a bit of magic.

2.1.1 Automatic creation

Indeed, even without having to explicitly create reverse relations (e.g by providing models to the `GM2MField` constructor), they are automatically created when an instance of a yet unknown model is added. This means that after having added `movie` to a `User`'s `preferred_videos`, you can do:

```
>>> [u.name for u in movie.user_set]
['Me']
```

However, it is important to remember that if no instance of a model has ever been added to the set, retrieving the `<modelname_set>` will raise an `AttributeError`:

```
>>> class Opera(Video):
>>>     pass
>>>
>>> bartered_bride = Opera.objects.create(title="The Bartered Bride")
>>> [u.name for u in bartered_bride.user_set]
AttributeError: 'Opera' object has no attribute 'user_set'
```

Indeed, the `GM2MField` has no idea what relation it is expected to create until you provide it with a minimum of information.

Warning: If automatic relations have been added during a session, be aware that they will not necessarily be available in another session. If you restart your server, for example, the automatically created relations will be lost. Read on to find how to tackle this issue.

2.1.2 Manual creation

If you want some reverse relations to be created before any instance is added, so that retrieving the `<modelname_set>` attribute never raises an exception, it is possible to explicitly provide some models to the `GM2MField` constructor. You may use model names (`'app.Model'` or `'Model'` if you're in the same module) if necessary to avoid circular references.

Let's say that instead of:

```
>>> class User(models.Model):
>>>     preferred_videos = GM2MField()
```

We actually write:

```
>>> class User(models.Model):
>>>     preferred_videos = GM2MField(Movie, 'Opera')
```

Then the reverse relations from `Movie` and `Opera` are created when the model classes are created and they are available even if no instance has been added yet:

```
>>> [u.name for u in bartered_bride.user_set]
[]
```

Note that providing models to `GM2MField` does not prevent you from adding instances from other models. You can still add instances from other models, and the relation will be created. Providing a list of models will only create reverse relations by default, nothing more.

2.1.3 Manual creation on existing model

If you need to add relations afterwards, or if the `GM2MField` is defined in a third-party library you cannot or do not want to patch, you can use the `GM2MField`'s `add_relation` method.

Suppose we could not amend our `User` class to add the reverse relations to `Movie` and `Opera` by providing arguments to the `GM2MField` constructor, this would have exactly the same effect:

```
>>> User.preferred_videos.add_relation(Movie)
>>> User.preferred_videos.add_relation('Opera')
```

As shown, you can also use model names (`app.Model`) with `add_relation`.

2.1.4 Operations and queries on reverse relations

The reverse relations provide you with the full set of operations that normal Django reverse relation exposes: `add`, `remove` and `clear`. `set` is also available from version 0.4.2.

A reverse relation also enables you to use lookup chains in your queries:

```
>>> jack = User.objects.create(name='Jack')
>>> jack.preferred_videos.add(bartered_bride)
>>> [o.name for o in Opera.objects.filter(user__name='Jack')]
['The Bartered Bride']
```

2.1.5 Related models lookup

From version 0.4.3 onwards, you can access all the models related to a `GM2MField` using the `get_related_models` method, that takes an `include_auto` optional argument if you want to include the automatically created models:

```
>>> User.preferred_videos.get_related_models()
[<class 'Movie'>, <class 'Opera'>]
>>>
>>> User.preferred_videos.get_related_models(include_auto=True)
[<class 'Movie'>, <class 'Opera'>, <class 'Documentary'>]
```

Indeed, in that example, `Movie`, `Opera` and `Theater` have been added to `preferred_videos`, while `Documentary` has only been automatically added with the addition of `citizenfour` to `me`'s preferred videos (at the top of the page).

2.2 Deletion

By default, when a source or target model instance is deleted, all relations linking this instance are deleted. It is possible to change this behavior with the `on_delete`, `on_delete_src` and `on_delete_tgt` keyword arguments when creating the `GM2MField`:

```
>>> from gm2m.deletion import DO_NOTHING
>>>
>>> class User(models.Model):
>>>     preferred_videos = GM2MField(Movie, 'Documentary',
>>>                                  on_delete=DO_NOTHING)
```

If you only want this behaviour on one side of the relationship (e.g. on the source model side), use `on_delete_src` or `on_delete_tgt`:

```
>>> class User(models.Model):
>>>    preferred_videos = GM2MField(Movie, 'Documentary',
>>>                                 on_delete_src=DO_NOTHING)
```

`on_delete_src` and `on_delete_tgt` override `on_delete`.

Several deletion functions are available:

CASCADE [default] The relation is deleted with the instance it is related to. The database remains consistent, no `ForeignKey` nor `GenericForeignKey` can point to a non-existent object after the operation.

DO_NOTHING The relation is not deleted with the instance it is related to. It is your responsibility to ensure that the database remains consistent after the deletion operation.

CASCADE_SIGNAL Same as CASCADE but sends the `deleting` signal (see *Signals* below).

CASCADE_SIGNAL_VETO Sends a `deleting` signal, and if no receiver vetoes the deletion by returning `True` or a Truthy value, calls CASCADE. Be careful using this one as when the deletion is vetoed, the database is left in an inconsistent state.

DO_NOTHING_SIGNAL Same as DO_NOTHING but sends a `deleting` signal.

2.3 Signals

The signals listed below can be imported from the `gm2m.signals` module.

deleting Sent when source model (= where the `GM2MField` is defined) instances are deleted. The `sender` is the `GM2MField` instance. The receivers take 2 keyword arguments:

- `del_objs`, an iterable containing the objects being deleted in the first place
- `rel_objs`, an iterable containing the objects related to the objects in `del_objs`, and that are to be deleted if cascade deletion is enabled

This signal can be used to customize the behaviour when deleting a source or target instance.

2.4 Prefetching

Prefetching works exactly the same way as with django `ManyToManyField`:

```
>>> User.objects.all().prefetch_related('preferred_videos')
```

will, in a minimum number of queries, prefetch all the videos in all the users' `preferred_video` lists.

2.5 Through models

Custom through models are also supported. The minimum requirements for through model classes are:

- one `ForeignKey` to the source model
- one `GenericForeignKey` with its `ForeignKey` and `CharField`

For example:

```
>>> class User(models.Model):
>>>     preferred_videos = GM2MField(through='PreferredVideos')
>>>
>>> class PreferredVideos(models.Model):
>>>     user = models.ForeignKey(User)
>>>     video = GenericForeignKey(ct_field='video_ct', fk_field='video_fk')
>>>     video_ct = models.ForeignKey(ContentType)
>>>     video_fk = models.CharField(max_length=255)
>>>
>>>     ... any relevant field (e.g. date added)
```

If there is only one ForeignKey to the source model (User in the above example) and only one GenericForeignKey in the target model, they will automatically be used for the relationship. Otherwise, if there are more of them, you must provide a `through_fields` argument (a list or tuple of 2 to 4 field names) to the `GM2MField` constructor.

2.6 GM2MField constructor's other parameters

In addition to the specific `on_delete*` and `through` / `through_fields` parameters, you can use the following optional keyword arguments when defining a `GM2MField`. For the sake of consistency, they have the same signification as in Django's `ManyToManyField` and `GenericForeignKey`.

verbose_name A human-readable name for the field. Defaults to a munged version of the model class name.

db_table The name of the database table to use for the automatically created through model. Defaults to `'<app_label>_<model_name>'`.

db_constraint Controls whether or not a constraint should be created in the database for the internal foreign keys when the through model is automatically created. Defaults to `True`.

for_concrete_model If set to `False`, the field will be able to reference proxy models. Defaults to `True`.

related_name The name that will be used for the relation from a related object back to this one. The same related name is used for all the related models. Defaults to `'<src_model_name>_set'`.

related_query_name The name to use for the reverse filter name from the target model. Defaults to the value of `related_name` or the model name.

pk_maxlength This is useful when using an automatically created intermediate model, to specify the length of the `CharField` used to store primary keys in the `GenericForeignKey`. Indeed, the default value of 16 characters may not be sufficient to accomodate certain large foreign key values (e.g. UUIDs). Use `None` if you don't want any limitation (this may cause performance issues, though). Defaults to `16`.

2.7 Migrations

`django-gm2m` fully supports Django migrations.

When generating migrations for an app using `GM2MField`, do not be surprised to see a `through_fields` keyword argument (as a list containing 4 field names) in the migration even if you did not provide it when creating the `GM2MField` in your model. This is necessary for django's migrations system to keep track of the arguments assignment and build accurate model representations from the migrations.

2.8 System checks

`django-gm2m` adds a few system checks, derived from built-in django checks for related fields and many to many fields. Here are the errors they may raise, with the builtin counterpart between brackets:

gm2m.E001 [fields.E330] GM2MFields cannot be unique

gm2m.E101 [fields.E331] Field specifies a many-to-many relation through model which has not been installed

gm2m.E102 [fields.E333] The model used as an intermediate model does not have a foreign key to the source model

gm2m.E103 [fields.E334] The model used as an intermediate model has more than one foreign key to the source model, which is ambiguous (the one that is used is the first declared in the model)

gm2m.E104 [fields.E333] The model used as an intermediate model does not have a generic foreign key

gm2m.E105 [fields.E334] The model used as an intermediate model has more than one generic foreign key, which is ambiguous (the one that is used is the first declared in the model).

gm2m.E106 [fields.E337] The field specifies `through_fields` but does not provide the names of the two link fields that should be used for the relation through model

gm2m.E107 [fields.E338] The model used as an intermediate model does not have the field specified in `through_fields`

gm2m.E108 [fields.E339] The field specified in `through_fields` is not a foreign key to the source model

gm2m.E109 [fields.E338] The model used as an intermediate model does not have the generic foreign key field specified in `through_fields`

gm2m.E110 [fields.E339] The field specified in `through_fields` is not a generic foreign key

gm2m.E201 [fieldsE301] Field defines a relation with a model that has been swapped out

gm2m.E202 [fields.E302] Reverse accessor for the field clashes with a field from the target model

gm2m.E203 [fields.E303] Reverse query name for the field clashes with a field from the target model

gm2m.E204 [fields.E304] Reverse accessor for the field clashes with reverse accessor from another field

gm2m.E205 [fields.E305] Reverse accessor for the field clashes with reverse query name from another field

2.9 Future improvements

- Add Django admin and possibly `limit_choices_to` support

CHAPTER 3

Warnings

3.1 Form field and django admin

Unlike django's `ManyToManyField`, `GM2MField` has no default associated form field. This may be added in the future, but for now a warning is raised when a `ModelForm` attempts to automatically create a field for `GM2MField`.

This warning is also raised when automatically creating an admin form for a model featuring a `GM2MField`.

3.2 (De)Serialization

Since version 0.4.2, `django-gm2m` supports serialization and deserialisation to and from fixture files (JSON, XML, YAML ...) using the `dumpdata` and `loaddata` management commands.

3.2.1 `dumpdata` and natural keys

As you probably already know, `django-gm2m` relies on `django.contrib.contenttypes` and needs to link `ContentType` objects. If you use the `dumpdata` command without excluding the `contenttypes` app and with standard primary/foreign keys, the data will contain dumped `ContentType` objects which will be referenced by their standard primary key (an integer).

When you'll attempt to load that data using `loaddata`, Django will at the same attempt to recreate the needed `ContentType` objects, which primary keys may not be consistent with your data, therefore raising a fixture loading error.

To avoid that, it is advised to use the `dumpdata` command with the following options in a project that makes use of `django-gm2m`:

- `--natural-primary --natural-foreign` to use natural keys instead of actual primary keys in the dumped data (the natural key for a contenttype is, for example, `'app_name.modelname'`)
- `-e contenttypes` to exclude the `ContentType` objects from the dumped data. These objects are automatically recreated by django anyway

See this StackOverflow question and answers for more details.

3.2.2 Custom serializers

When a project using `django-gm2m` is initialized, the default django serializers (namely `json`, `xml`, `yaml`) are overridden by specific serializers that have been tuned to work with `GM2MField`.

This means that in case you have custom serializers in your project or app, you will need to derive them from `gm2m.serializers.*.Serializer` instead of `django.core.serializers.*.Serializer` (same for `Deserializer`). If you don't do that, (de)serialization of `GM2MFields` will not work.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search