Model Object Mapper Documentation

Release 0.0.4-alpha

Model Object Mapper

Jun 05, 2021

USAGE

Welcome to the Model Object Mapper Documentation page!

Model Object Mapper, or *MOM* for short, is a Django Management Utility for statically creating and updating database entries. It supports foreign fields and can work with Django apps without requiring any modification.

- Source Code
- Issue Tracker
- Releases
- PyPI

CHAPTER

ONE

OVERVIEW

Let's assume that you have the following Django database model:

Listing 1: myapp/models.py

```
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=100, )
    date = models.DateTimeField()
    slug = models.SlugField(unique=True, )
```

In your *mom.yaml* file, you can refer to this model like this:

Listing 2: mom_data/mom.yaml

```
mom:
  map:
    post:
      model: myapp.models.Post
      lookupField: slug
```

The `post` key in which we have `model` and `lookupField` keys is the name prefix for files that MOM will load and use.

As an example, we will create a serialization file that will target the `myapp.models.Post` model:

Listing 3: mom_data/post.first-post.yaml

```
field:
  title: My Awesome Title
  date: 2021-06-25 13:00
```

As you can see, we don't have the `slug` field here. This is because `lookupField` sets what the root name is going to be used for. In that case, `first-post` will be the value of the `slug` field.

Now, you can run the MOM command:

```
$ ./manage.py mom
```

This will create the database row or update the existing row after you change the any of the values in the file and run the MOM command like above.

CHAPTER

TWO

INSTALLATION

First, install the latest version of MOM:

```
$ python3 -m pip install django-mom
```

Next, add `django-mom` to the `INSTALLED_APPS` list in your Django app's **settings.py** file:

Listing 1: myapp/settings.py

```
INSTALLED_APPS = [
    'django.contrib.staticfiles',
    'django_mom',
]
```

Now you are ready to use MOM in your app. To test it, (after setting up Django and making sure it works) you can run:

```
$ ./manage.py mom
```

This should output something similar to this: `Couldn't open 'mom_data/mom.yaml' file.` which is the expected output if you haven't worked with MOM before.

Next, we will create the *mom_data* folder which is the default folder where MOM related files are put. It should be in the root folder of your Django app. You can customize the path by setting `MOM_FOLDER` to your liking.

You can also override the `MOM_FILE` variable that is used for pattern matching when looking for MOM related files. It is set to *mom.yaml* by default.

With custom paths, your settings file will look like this:

Listing 2: myapp/settings.py

```
MOM_FOLDER = 'mom_data'
MOM_FILE = 'mom.yaml'
```

CHAPTER

THREE

QUICKSTART GUIDE

3.1 Main MOM File

Now that you have created the `mom_data` folder, you can create the main MOM file in which we will define the database models that we are going to use.

The main MOM file is where we define the models and their relationships with other models. If you don't plan to use related fields, e.g. foreign keys, the latter will not be a concern to us.

Now let's take a look at this MOM file:

Listing 1: mom_data/mom.yaml

```yaml
mom:
  map:
    user:
      model: myapp.models.User
      lookupField: username
    topic:
      model: myapp.models.Topic
      lookupField: slug
```

In this configuration, the model `myapp.models.User` is tied to the `user` key and `myapp.models.Topic` to `topic`. MOM will look for patterns that match these two and sync the matching files with the database using the models that they target.

MOM will accept files nested in folders. In other words, you can structure them in 3 different ways depending on your preference. To illustrate that, let's assume that we have a `myapp.models.User` data that we are going to serialize with MOM:

```
.
└── mom_data
    ├── mom.yaml
    ├── user
    │   ├── velitasali
    │   │   └── mom.yaml
    │   └── velitasali.mom.yaml
    └── user.velitasali.mom.yaml

 3 directories, 4 files
```

Ignoring the `mom_data/mom.yaml`, here we can see the files `mom_data/user.velitasali.mom.yaml`, `mom_data/user/velitasali.mom.yaml`, and `mom_data/user/velitasali/mom.yaml` point to the same row on the database.

Note: *mom.yaml*, as previously mentioned, is set by the `MOM_FILE` variable in the `myapp/settings.py` file. By changing it, you will also be changing which files are considered to be MOM files.

velitasali is used as a value to the `username` field of `myapp.models.User` as defined in the `mom_data/mom.yaml` file.

Note: `lookupField` should be a unique field on the database. If more than one row appears when queried, MOM will fail.

3.2 Serialization Files

Continuing from the `myapp.models.User` example, let's see the insides of the file `mom_data/user.velitasali.mom.yaml` which contains the serialization data and does all the magic.

First, let's have a look at the model itself:

Listing 2: myapp/models.py

```
from django.db import models

class User(models.Model):
    username = models.SlugField(primary_key=True, )
    email = models.EmailField(max_length=100, )
    password = models.CharField(max_length=100, )
    full_name = models.CharField(max_length=64, )
```

A corresponding serialization file would look like this:

Listing 3: mom_data/user.velitasali.mom.yaml

```
field:
  email: velitasali@site.com
  password: unhackable_1234
  full_name: Veli Tasalı
```

As previously discussed, the value for the `username` field is coming from the file name.

3.2.1 Files As Values

Also, if we wanted to fill one of the values using a file, we could do that as well.

Listing 4: mom_data/user.velitasali.mom.yaml

```
field:
  # ...
  full_name file: name.txt
```

In this example, MOM will look for the file `mom_data/name.txt` and load the value from it. The location of the file will depend on where the serialization file is. For instance, if the serialization file were located at `mom_data/user/velitasali.mom.yml`, MOM would look for the file `name.txt` in the `mom_data/user/` folder.

3.2.2 Working with FileField and Derivatives

Another neat feature of MOM is the ability to work with `FileField` type of fields, emulating a file uploading behaviour.

You can specify a file that MOM will observe for changes and replace when there is a change. The file will be copied to the storage that the field uses and will always be up-to-date.

To enable this feature, append `djangofile` option after the field name and give the file location as the value.

As an example, let's create a model that has a `FileField` field:

Listing 5: myapp/models.py

```
class UserFile(models.Model):
    slug = models.SlugField(primary_key=True, )
    file = models.FileField(upload_to='media', )
```

Next, create a Main MOM file that uses that model:

Listing 6: mom_data/mom.yaml

```
mom:
  map:
    file:
      model: myapp.models.UserFile
      lookupField: slug
```

Finally, create a serialization file that uses `djangofile` option:

Listing 7: mom_data/file.myfile.mom.yaml

```
field:
  file djangofile: myfile.txt
```

With this, MOM will observe `myfile.txt` for change and replace it when needed.

As you can see, this is similar to `file` option with the difference being `djangofile` passing a file to the field and the other reading the file contents and passing it as the value.

3.3 Relational Fields

MOM supports relational fields such as Foreign Keys or ManyToMany fields. The first and easiest way to work with them is by providing their unique field name and value in the serialization file, but first, let's add a relational field to our `myapp.models.User` model:

Listing 8: myapp/models.py

```
# ...

class Rank(models.Model):
    name = models.CharField(primary_key=True, max_length=100, )

class User(models.Model):
    # ...
    rank = models.ForeignKey(Rank, on_delete=models.PROTECT)
```

Now let's assume that we already have two rows in the *Rank* table which are:

1. Rank(name='user')
2. Rank(name='admin')

To set the one for our user *User(username='velitasali')*, we can:

Listing 9: mom_data/user.velitasali.mom.yaml

```
1 field:
2   rank:
3     name: admin
4   email: velitasali@site.com
5   password: unhackable_1234
6   full_name: Veli Tasalı
```

As you can see in the lines 2 and 3, we are providing the *rank* field which is a related *myapp.models.Rank* model just by providing its lookup field and value.

3.3.1 Creating Dependencies

You can also create a *rank* with MOM, which is very similar to what we have been doing so far. To do so, we will first add the `myapp.models.Rank` to the main MOM file:

Listing 10: mom_data/mom.yaml

```
1 mom:
2   map:
3     user:
4       # ...
5     rank:
6       model: myapp.models.Rank
7       lookupField: name
```

With this change MOM will also look for `rank` key alongside `user` key. Also, MOM will wait until the `rank` it is looking for exists. This is the general behavior of MOM where if a given relational field doesn't exist, it skips that serialization file momentarily or fails if the given field doesn't come up at some point.

Now we can create our `myapp.models.Rank` serialization file to create a new row in the database which we will then feed to `mom_data/user.velitasali.mom.yaml`. First, create a new file:

Listing 11: mom_data/rank.moderator.mom.yaml

```
field:
```

As you can see, the `field` key is empty. This is because the only field that the model needs, `name`, is provided by the file name. There is a shorter way to handle scenarios like this, but let's first update the `user` file before we use the shortcuts.

Listing 12: mom_data/user.velitasali.mom.yaml

```
1 field:
2   rank:
3     name: moderator
4   # ...
```

Now you can run:

```
$ ./manage.py mom
```

MOM will generate the new `rank` entry and assign it to the `user`.

3.3.2 Implicitly Creating Shared Dependencies

You can also create a `rank` implicitly without needing an extra file. To do so, we will first rewrite our main MOM file `mom_data/mom.yaml`:

Listing 13: mom_data/mom.yaml

```
mom:
  map:
    user:
      model: myapp.models.User
      lookupField: username
  remap:
    myapp.models.Rank:
      lookupField:
        - name
      ownership: shared
```

By default, MOM does not create or update relational fields unless you explicitly allow it to. By modifying the main MOM file like above, we are telling MOM to generate any `myapp.models.Topic` related field with *shared* `ownership`.

3.3.3 Implicitly Creating Single Dependencies

Now since `slug` is the only field that `myapp.models.Rank` requires, this will be enough for MOM to generate the row when it doesn't exist. However, by using *shared* `ownership`, we tell it not to update an existing row but to create a new one or switch to an existing one when we supply a newer value to an existing `user` entry. This is useful when `rank` is used or shared between more than one row in the database.

You can also use *single* `ownership` which tells MOM that the value is owned by `user` or any other database model that uses the `myapp.models.Rank` model. In that case, altering the value in `user` will alter `rank` itself if `user` already have had value. To give an example, let's assume that we are dealing with an empty database and this is the first time we are running the app:

Listing 14: mom_data/mom.yaml

```
mom:
  # ...
  remap:
    myapp.models.Rank:
      # ...
      ownership: single
```

Listing 15: mom_data/user.velitasali.mom.yaml

```
field:
  rank:
    name: user
  email: velitasali@site.com
```

(continues on next page)

(continued from previous page)

```
  password: unhackable_1234
  full_name: Veli Tasalı
```

This will create a new `rank` entry. Now let's assume that we are changing the `rank.name` to something else and re-run MOM:

Listing 16: mom_data/user.velitasali.mom.yaml

```
field:
  rank:
    name: developer
  # ...
```

What happens is the `rank` entry that we had is now *Rank(name=developer)*. This is because the `user` entry already had a `rank` value and, with *single* `ownership`, MOM altered it according to the change we have made.

Now that we have understood this part, we can get started with complex relational fields.

3.3.4 Using Multiple Lookup Fields

As you might have realized, MOM expects you to provide it with a unique field and value. However, in some cases, you may want to use multiple fields to locate a row. For foreign fields, this is possible. In a previous example, we have provided a **list** for `lookupField` under the `remap` key. Now let's look at this example:

Listing 17: mom_data/mom.yaml

```
mom:
  # ...
  remap:
    myapp.models.Rank:
      lookupField:
        - name
        - power
      lookupFieldOptional:
        - effect
      ownership: single
```

With this configuration, whenever MOM sees a related field that uses the `myapp.models.Rank` model, it will expect `name` and `power` fields to be present so that it can use them to look up for an existing row. Also, `lookupFieldOptional` sets which fields may also be optionally used when they are present. In that case, when `effect` field is present, MOM will use it as a lookup field.

Now we can update our `myapp.models.Rank` model according to this scenario:

Listing 18: myapp/models.py

```
# ...

class Rank(models.Model):
    name = models.CharField(max_length=100, )
    power = models.IntegerField()
    effect = models.IntegerField()
    subtitle = models.CharField(max_length=100, )
```

Now let's create a `user` example:

Listing 19: mom_data/user.velitasali.mom.yaml

```yaml
field:
  rank:
    name: user
    power: 70
    effect: 50
    subtitle: Just a user
  # ...
```

As you can see, we have provided every field that a `myapp.models.Rank` model needs since we will also be creating it. However, when looking up for an existing row, MOM will use only the `name`, `power`, and `effect` fields, leaving the `subtitle` to be used when altering the row only.

Note: MOM doesn't modify or check unreferenced fields. If `subtitle` had a default value and we hadn't mentioned in the `mom_data/user.velitasali.mom.yaml` file, Django would handle it for us.

3.3.5 ManyToMany Fields

You can also use ManyToMany fields by only by changing how you are referring to them. For instance, if the `rank` field of `user` were a ManyToMany field, you could do this:

Listing 20: mom_data/user.velitasali.mom.yaml

```yaml
field:
  rank:
    - name: user
      power: 70
      effect: 50
      subtitle: Just a user
    - name: admin
      power: 100
      effect: 100
      subtitle: Also an admin
  # ...
```

Although `rank` being a ManyToMany field doesn't make sense, you can see the difference between a ForeignKey and ManyToMany field.

Also, since `myapp.models.Rank` has *single* ownership, removing one of the fields from the `rank` field of `user` would mean deleting it altogether.

3.3.6 Deeply Nested Fields

MOM can work with nested fields deeper than what we have seen so far. You can use them as lookup fields as well. Let's say we have a third model:

Listing 21: myapp/models.py

```python
class Rank(models.Model):
    name = models.CharField(max_length=100, )
```

(continues on next page)

(continued from previous page)

```
class User(models.Model):
    username = models.SlugField(primary_key=True, )
    email = models.EmailField(max_length=100, )
    password = models.CharField(max_length=100, )
    full_name = models.CharField(max_length=64, )
    rank = models.ForeignKey(Rank, on_delete=models.PROTECT)

class Topic(models.Model):
    user = models.ForeignKey(User, on_delete=models.PROTECT)
    slug = models.SlugField(primary_key=True, )
```

Listing 22: mom_data/mom.yaml

```
mom:
  map:
    topic:
      model: myapp.models.Topic
      lookupField: slug
```

This time we didn't define a `remap` key because, by default, MOM considers all of the fields as lookup fields when there is no remapping defined. Also, again by default, `ownership` is set to *none*. This means MOM will not try to create a relational row when there is no `remap` data specifying `ownership` and lookup fields for a given model.

Now we can use a relational field as a lookup field:

Listing 23: mom_data/topic.mytopic.mom.yaml

```
field:
  user:
    rank:
      name: user
```

In this example, MOM will use `rank__name` as the lookup field to find the row for the `rank` field.

3.3.7 Implicit Lookup Field Passing

There is also a shortcut to doing all of this as if we are providing the value for a non-relational field. For that, we are going to define some `remap` values telling MOM to what to default to when working with a scenario like this:

Listing 24: mom_data/mom.yaml

```
mom:
  # ..
  remap:
    myapp.models.User:
      lookupField:
        - rank
      ownership: none
    myapp.models.Rank:
      lookupField:
        - name
      ownership: none
```

Now that we have defined the default lookup fields, we can optimize the previous serialization file like this:

Listing 25: mom_data/topic.mytopic.mom.yaml

```
field:
  user: user
```

What happens is MOM knows it should have received a *dict* or *list* for the `user` field since it is a relational field. Because that is not the case, it checks if the default field for the `myapp.models.User` is defined and then it resolves to `{'rank': 'user'}`. Then the same thing happens for the `rank` which then becomes `{'name': user}`. What we finally have is:

Listing 26: mom_data/topic.mytopic.mom.yaml

```
field:
  user:
    rank:
      name: user
```

As you can see, this is the same as what we previously had but without the boilerplate.

Note: This only works if there is only one field for the given model in the `lookupField` list.
