
Model Mommy Documentation

Release 1.2.1

Lucas Simon Rodrigues Magalhaes

December 08, 2015

1	Contributing to Model Mommy	3
2	Install	5
3	Contributing	7
4	Inspiration	9
5	Doubts? Loved it? Hated it? Suggestions?	11
5.1	Basic Usage	11
5.2	How mommy behaves?	15
5.3	Recipes	17
5.4	Deprecation Warnings	22
5.5	Known Issues	22
5.6	Extensions	22
6	Indices and tables	23

Model-mommy offers you a smart way to create fixtures for testing in Django. With a simple and powerful API you can create many objects with a single line of code.

Contributing to Model Mommy

As an open source project, Model Mommy welcomes contributions of many forms

Examples of contributions include:

- Code Patches
- Documentation improvements
- Bug reports

Install

Run the command above

```
pip install model_mommy
```

Contributing

1. Prepare a virtual environment.

```
pip install virtualenvwrapper
mkvirtualenv model_mommy --no-site-packages --distribute
```

2. Install the requirements.

```
pip install -r dev_requirements.txt
```

3. Run the tests.

```
make test
```

Inspiration

Model-mommy was inspired by many great open source software like ruby's ObjectDaddy and FactoryGirl.

Doubts? Loved it? Hated it? Suggestions?

Join our mailing list for support, development and ideas!

- <https://groups.google.com/group/model-mommy>

Contents:

5.1 Basic Usage

Let's say you have an app **family** with a model like this:

File: model.py

```
class Kid(models.Model):
    """
    Model class Kid of family app
    """
    happy = models.BooleanField()
    name = models.CharField(max_length=30)
    age = models.IntegerField()
    bio = models.TextField()
    wanted_games_qtd = models.BigIntegerField()
    birthday = models.DateField()
    appointment = models.DateTimeField()
```

To create a persisted instance, just call Mommy:

File: test_model.py

```
# -*- coding:utf-8 -*-

#Core Django imports
from django.test import TestCase

#Third-party app imports
from model_mommy import mommy
from model_mommy.recipe import Recipe, foreign_key

# Relative imports of the 'app-name' package
from .models import Kid

class KidTestModel(TestCase):
    """
```

```
Class to test the model
Kid
"""

def setUp(self):
    """
    Set up all the tests
    """
    self.kid = mommy.make(Kid)
```

No need to pass attributes every damn time.

Importing every model over and over again is boring. So let Mommy import them for you:

```
from model_mommy import mommy

# 1st form: app_label.model_name
kid = mommy.make('family.Kid')

# 2nd form: model_name
dog = mommy.make('Dog')
```

Note: You can only use the 2nd form on unique model names. If you have an app family with a Dog, and an app farm with a Dog, you must use the `app_label.model_name` form.

Note: `model_name` is case insensitive.

5.1.1 Model Relationships

Mommy also handles relationships. Say the kid has a dog:

File: `model.py`

```
class Kid(models.Model):
    """
    Model class Kid of family app
    """
    happy = models.BooleanField()
    name = models.CharField(max_length=30)
    age = models.IntegerField()
    bio = models.TextField()
    wanted_games_qtd = models.BigIntegerField()
    birthday = models.DateField()
    appointment = models.DateTimeField()

    class Meta:
        verbose_name = _(u'Kid')
        verbose_name_plural = _(u'Kids')

    def __unicode__(self):
        """
        Return the name of kid
        """
        return u'%s' % (self.name)

class Dog(models.Model):
```



```

"""
Model class Dog of family app
"""
owner = models.ForeignKey('Kid')

```

when you ask Mommy:

File: test_model.py

```

# -*- coding:utf-8 -*-

#Core Django imports
from django.test import TestCase

#Third-party app imports
from model_mommy import mommy
from model_mommy.recipe import Recipe, foreign_key

# Relative imports of the 'app-name' package

class DogTestModel(TestCase):
    """
    Class to test the model
    Dog
    """

    def setUp(self):
        """
        Set up all the tests
        """
        self.rex = mommy.make('family.Dog')

```

She will also create the Kid, automatically. **NOTE: ForeignKeys and OneToOneFields** Since Django 1.8, ForeignKey and OneToOne fields don't accept unpersisted model instances anymore. This means if you do:

```
mommy.prepare('family.Dog')
```

You'll end with a persisted "Kid" instance.

5.1.2 M2M Relationships

File: test_model.py

```

# -*- coding:utf-8 -*-

#Core Django imports
from django.test import TestCase

#Third-party app imports
from model_mommy import mommy
from model_mommy.recipe import Recipe, foreign_key

# Relative imports of the 'app-name' package

class DogTestModel(TestCase):
    """
    Class to test the model
    Dog
    """

```

```
def setUp(self):
    """
    Set up all the tests
    """
    self.rex = mommy.make('family.Dog', make_m2m=True)
```

5.1.3 Defining some attributes

Of course it's possible to explicitly set values for attributes.

File: test_model.py

```
# -*- coding:utf-8 -*-

#Core Django imports
from django.test import TestCase

#Third-party app imports
from model_mommy import mommy
from model_mommy.recipe import Recipe, foreign_key

# Relative imports of the 'app-name' package
from .models import Kid

class KidTestModel(TestCase):
    """
    Class to test the model
    Kid
    """

    def setUp(self):
        """
        Set up all the tests
        """
        self.kid = mommy.make(
            Kid,
            age=3
        )

        self.another_kid = mommy.make(
            'family.Kid',
            age=6
        )
```

Related objects attributes are also reachable:

File: test_model.py

```
# -*- coding:utf-8 -*-

#Core Django imports
from django.test import TestCase

#Third-party app imports
from model_mommy import mommy
from model_mommy.recipe import Recipe, foreign_key
```

```
# Relative imports of the 'app-name' package
from .models import Dog

class DogTestModel(TestCase):
    """
    Class to test the model
    Dog
    """

    def setUp(self):
        """
        Set up all the tests
        """

        self.bobs_dog = mommy.make(
            'family.Dog',
            owner__name='Bob'
        )
```

5.1.4 Non persistent objects

If you don't need a persisted object, *Mommy* can handle this for you as well:

```
from model_mommy import mommy

kid = mommy.prepare('family.Kid')
```

It works like *make*, but it doesn't persist the instance.

5.1.5 More than one instance

If you need to create more than one instance of the model, you can use the *_quantity* parameter for it:

```
from model_mommy import mommy

kids = mommy.make('family.Kid', _quantity=3)
assert len(kids) == 3
```

It also works with *prepare*:

```
from model_mommy import mommy

kids = mommy.prepare('family.Kid', _quantity=3)
assert len(kids) == 3
```

5.2 How mommy behaves?

By default, *model-mommy* skips fields with *null=True* or *blank=True*. Also if a field has a *default* value, it will be used.

You can override this behavior by:

1. Explicitly defining values

```
# from "Basic Usage" page, assume all fields either null=True or blank=True
from .models import Kid
from model_mommy import mommy

kid = mommy.make(Kid, happy=True, bio='Happy kid')
```

2. Passing `_fill_optional` with a list of fields to fill with random data

```
kid = mommy.make(Kid, _fill_optional=['happy', 'bio'])
```

3. Passing `_fill_optional=True` to fill all fields with random data

```
kid = mommy.make(Kid, _fill_optional=True)
```

5.2.1 When shouldn't you let mommy generate things for you?

If you have fields with special validation, you should set their values by yourself.

Model-mommy should handle fields that:

1. don't matter for the test you're writing;
2. don't require special validation (like unique, etc);
3. are required to create the object.

5.2.2 Currently supported fields

- BooleanField, IntegerField, BigIntegerField, SmallIntegerField, PositiveIntegerField, PositiveSmallIntegerField, FloatField, DecimalField
- CharField, TextField, BinaryField, SlugField, URLField, EmailField, IPAddressField, GenericIPAddressField
- ForeignKey, OneToOneField, ManyToManyField (even with through model)
- DateField, DateTimeField, TimeField
- FileField, ImageField

5.2.3 Custom fields

Model-mommy allows you to define generators methods for your custom fields or overrides its default generators. This could be achieved by specifying a dict on settings with keys defining the fields and values the generator functions. Both can be the real python objects imported in settings or just specified as import path string.

Examples:

```
# on your settings.py file:
def gen_func():
    return 'value'

MOMMY_CUSTOM_FIELDS_GEN = {
    'test.generic.fields.CustomField': gen_func,
}
```

```
# in the module code.path:
def gen_func():
    return 'value'

# in your settings.py file:
MOMMY_CUSTOM_FIELDS_GEN = {
    'test.generic.fields.CustomField': 'code.path.gen_func',
}
```

5.3 Recipes

If you're not comfortable with random data or even you just want to improve the semantics of the generated data, there's hope for you.

You can define a recipe, which is a set of rules to generate data for your models. Create a module called `mommy_recipes.py` at your app's root directory:

```
fixtures/
migrations/
templates/
tests/
__init__.py
admin.py
managers.py
models.py
mommy_recipes.py
urls.py
views.py
```

File: `mommy_recipes.py`

```
from model_mommy.recipe import Recipe
from family.models import Person

person = Recipe(
    Person,
    name = 'John Doe',
    nickname = 'joe',
    age = 18,
    birthday = date.today(),
    appointment = datetime.now()
)
```

Note: You don't have to declare all the fields if you don't want to. Omitted fields will be generated automatically.

File: `test_model.py`

```
# -*- coding:utf-8 -*-

#Core Django imports
from django.test import TestCase

#Third-party app imports
from model_mommy import mommy
from model_mommy.recipe import Recipe, foreign_key
```



```

self.employee_recipe = Recipe(
    Employee,
    name=seq('Employee '),
    company=company_recipe.make()
)

def test_employee_list(self):
    self.employee_recipe.make(_quantity=3)
    # test stuff...

def test_employee_tasks(self):
    employee1 = self.employee_recipe.make()
    task_recipe = Recipe(Task, employee=employee1)
    task_recipe.make(status='done')
    task_recipe.make(due_date=datetime(2014, 1, 1))
    # test stuff...

```

5.3.1 Recipes with foreign keys

You can define *foreign_key* relations:

```

from model_mommy.recipe import Recipe, foreign_key
from family.models import Person, Dog

person = Recipe(Person,
    name = 'John Doe',
    nickname = 'joe',
    age = 18,
    birthday = date.today(),
    appointment = datetime.now()
)

dog = Recipe(Dog,
    breed = 'Pug',
    owner = foreign_key(person)
)

```

Notice that *person* is a *recipe*.

You may be thinking: “I can put the *Person* model instance directly in the *owner* field”. That’s not recommended.

Using the *foreign_key* is important for 2 reasons:

- Semantics. You’ll know that attribute is a foreign key when you’re reading;
- The associated instance will be created only when you call *make_recipe* and not during recipe definition;

You can also use *related*, when you want two or more models to share the same parent:

```

from model_mommy.recipe import related, Recipe

dog = Recipe(Dog,
    breed = 'Pug',
)
other_dog = Recipe(Dog,
    breed = 'Boxer',
)
person_with_three_dogs = Recipe(Person,

```

```
dog_set = related('dog', 'other_dog')
)
```

Note this will only work when calling `make_recipe` because the related manager requires the objects in the `related_set` to be persisted. That said, calling `prepare_recipe` the `related_set` will be empty.

If you want to set m2m relationship you can use `related` as well:

```
class Dog(models.Model):
    owner = models.ForeignKey('Person')
    breed = models.CharField(max_length=50)
    created = models.DateTimeField(auto_now_add=True)
    friends_with = models.ManyToManyField('Dog')

from model_mommy.recipe import related, Recipe

dog = Recipe(Dog,
    breed = 'Pug',
)

dog_with_friends = dog.extend(
    friends_with=related(dog, dog),
)
```

5.3.2 Recipes with callables

It's possible to use *callables* as recipe's attribute value.

```
from datetime import date
from model_mommy.recipe import Recipe
from family.models import Person

person = Recipe(Person,
    birthday = date.today,
)
```

When you call `make_recipe`, *Mommy* will set the attribute to the value returned by the callable.

5.3.3 Recipes with iterators

You can also use *iterators* (including *generators*) to provide multiple values to a recipe.

```
from itertools import cycle

colors = ['red', 'green', 'blue', 'yellow']
person = Recipe(Person,
    favorite_color = cycle(colors)
)
```

Mommy will use the next value in the *iterator* every time you create a model from the recipe.

5.3.4 Sequences in recipes

Sometimes, you have a field with an unique value and using `make` can cause random errors. Also, passing an attribute value just to avoid uniqueness validation problems can be tedious. To solve this you can define a sequence with `seq`


```

from model_mommy.recipe import Recipe, seq
from family.models import Person

person = Recipe(Person,
    name = seq('Joe'),
    age = seq(15)
)

p = mommy.make_recipe('myapp.person')
p.name
>>> 'Joe1'
p.age
>>> 16

p = mommy.make_recipe('myapp.person')
p.name
>>> 'Joe2'
p.age
>>> 17

```

This will append a counter to strings to avoid uniqueness problems and it will sum the counter with numerical values.

You can also provide an optional *increment_by* argument which will modify incrementing behaviour. This can be an integer, float, Decimal or timedelta.

```

from datetime import datetime, timedelta
from model_mommy.recipe import Recipe, seq
from family.models import Person

person = Recipe(Person,
    age = seq(15, increment_by=3)
    height_ft = seq(5.5, increment_by=.25)
    # assume today's date is 21/07/2014
    appointment = seq(datetime.date(2014, 7, 21), timedelta(days=1))
)

p = mommy.make_recipe('myapp.person')
p.age
>>> 18
p.height_ft
>>> 5.75
p.appointment
>>> datetime.date(2014, 7, 22)

p = mommy.make_recipe('myapp.person')
p.age
>>> 21
p.height_ft
>>> 6.0
p.appointment
>>> datetime.date(2014, 7, 23)

```

Note: If your Python's interpreter version is 2.6.x or earlier then *increment_by* is not available for you. *seq* will simply ignore this argument.

5.3.5 Overriding recipe definitions

Passing values when calling *make_recipe* or *prepare_recipe* will override the recipe rule.

```
from model_mommy import mommy

mommy.make_recipe('model_mommy.person', name='Peter Parker')
```

This is useful when you have to create multiple objects and you have some unique field, for instance.

5.3.6 Recipe inheritance

If you need to reuse and override existent recipe call *extend* method:

```
dog = Recipe(Dog,
             breed = 'Pug',
             owner = foreign_key(person)
)
extended_dog = dog.extend(
    breed = 'Super basset',
)
```

5.4 Deprecation Warnings

Because of the changes of *model_mommy*'s API, the following methods are deprecated and will be removed in one of the future releases:

- *mommy.make_one* -> should use the method *mommy.make* instead
- *mommy.prepare_one* -> should use the method *mommy.prepare* instead
- *mommy.make_many* -> should use the method *mommy.make* with the *_quantity* parameter instead
- *mommy.make_many_from_recipe* -> should use the method *mommy.make_recipe* with the *_quantity* parameter instead

5.5 Known Issues

5.5.1 django-taggit

Model-mommy identifies *django-taggit*'s *TaggableManager* as a normal Django field, which can lead to errors:

```
TypeError: <class 'taggit.managers.TaggableManager'> is not supported by mommy.
```

The fix for this is to set `blank=True` on your *TaggableManager*.

5.6 Extensions

5.6.1 GeoDjango

Works with it? This project has some custom generators for it: https://github.com/sigma-consultoria/mommy_spatial_generators

Indices and tables

- `genindex`
- `modindex`
- `search`