Model Bakery Documentation

Release 1.0.1

Lucas Simon Rodrigues Magalhaes

Oct 21, 2019

Contents

1 Contributing to Model Bakery **3**

2 Compatibility **5**

3 Install **7**

4 Contributing **9**

5 Doubts? Loved it? Hated it? Suggestions? **11**

5.1 Basic Usage . . . 11
5.1.1 Model Relationships . . . 12
5.1.2 M2M Relationships . . . 13
5.1.3 Explicit M2M Relationships . . . 13
5.1.4 Explicit values for fields . . . 13
5.1.5 Creating Files . . . 14
5.1.6 Non persistent objects . . . 14
5.1.7 More than one instance . . . 15
5.2 Recipes . . . 15
5.2.1 Recipes with foreign keys . . . 16
5.2.2 Recipes with callables . . . 17
5.2.3 Recipes with iterators . . . 18
5.2.4 Sequences in recipes . . . 18
5.2.5 Overriding recipe definitions . . . 19
5.2.6 Recipe inheritance . . . 20
5.3 How Model Bakery behaves? . . . 20
5.3.1 When shouldn't you let Baker generate things for you? . . . 20
5.3.2 Currently supported fields . . . 20
5.3.3 Custom fields . . . 21
5.3.4 Customizing Baker . . . 21
5.3.5 Save method custom parameters . . . 22
5.4 Migrating from Model Mommy . . . 22

Model Bakery 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.

Model Bakery is a rename of the legacy model_mommy's project.

CHAPTER 1

Contributing to Model Bakery

As an open source project, Model Bakery welcomes contributions of many forms. Examples of contributions include:

- Code Patches
- Documentation improvements
- Bug reports

CHAPTER 2

Compatibility

model_bakery supports Django >= 1.11 and Python >= 3.5

CHAPTER 3

Install

Install it with `pip`

```
$ pip install model_bakery
```

CHAPTER 4

Contributing

1. Prepare a virtual environment.

```
$ pip install virtualenvwrapper
$ mkvirtualenv model_bakery
```

2. Install the requirements.

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

3. Run the tests.

```
$ make test
```

CHAPTER 5

Doubts? Loved it? Hated it? Suggestions?

Feel free to open an issue for support, development or ideas!

- https://github.com/model-bakers/model_bakery

Contents:

5.1 Basic Usage

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

File: **models.py**

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

To create a persisted instance, just call Model Bakery:

File: **test_models.py**

```
#Core Django imports
from django.test import TestCase

#Third-party app imports
from model_bakery import baker
```

(continues on next page)

(continued from previous page)

```
from shop.models import Customer

class CustomerTestModel(TestCase):
    """
    Class to test the model Customer
    """

    def setUp(self):
        self.customer = baker.make(Customer)
```

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

```
from model_bakery import baker

# 1st form: app_label.model_name
customer = baker.make('shop.Customer')

# 2nd form: model_name
product = baker.make('Product')
```

Note: You can only use the 2nd form on unique model names. If you have an app shop with a Product, and an app stock with a Product, you must use the app_label.model_name form.

Note: model_name is case insensitive.

5.1.1 Model Relationships

Model Bakery also handles relationships. Let's say the customer has a purchase history:

File: **models.py**

```
class Customer(models.Model):
    """
    Model class Customer of shop 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 PurchaseHistory(models.Model):
    """
    Model class PurchaseHistory of shop app
    """
    customer = models.ForeignKey('Customer')
    products = models.ManyToManyField('Product')
    year = models.IntegerField()
```

You can use Model Bakery as:

```
from django.test import TestCase

from model_bakery import baker

class PurchaseHistoryTestModel(TestCase):

    def setUp(self):
        self.history = baker.make('shop.PurchaseHistory')
        print(self.history.customer)
```

It will also create the Customer, automagically.

NOTE: ForeignKeys and OneToOneFields - Since Django 1.8, ForeignKey and OneToOne fields don't accept unpersisted model instances anymore. This means that if you run:

```
baker.prepare('shop.PurchaseHistory')
```

You'll end up with a persisted "Customer" instance.

5.1.2 M2M Relationships

By default Model Bakery doesn't create related instances for many-to-many relationships. If you want them to be created, you have to turn it on as following:

```
from django.test import TestCase

from model_bakery import baker

class PurchaseHistoryTestModel(TestCase):

    def setUp(self):
        self.history = baker.make('shop.PurchaseHistory', make_m2m=True)
        print(self.history.products.count())
```

5.1.3 Explicit M2M Relationships

If you want to, you can prepare your own set of related object and pass it to Model Bakery. Here's an example:

```
products_set = baker.prepare(Product, _quantity=5)
history = baker.make(PurchaseHistory, products=products_set)
```

5.1.4 Explicit values for fields

By default, Model Bakery uses random values to populate the model's fields. But it's possible to explicitly set values for them as well.

```
from django.test import TestCase

from model_bakery import baker

class CustomerTestModel(TestCase):
```

(continues on next page)

(continued from previous page)

```
    def setUp(self):
        self.customer = baker.make(
            'shop.Customer',
            age=21
        )

        self.older_customer = baker.make(
            'shop.Customer',
            age=42
        )
```

Related objects fields are also reachable by their name or related names in a very similar way as Django does with field lookups:

```
from django.test import TestCase

from model_bakery import baker

class PurchaseHistoryTestModel(TestCase):

    def setUp(self):
        self.bob_history = baker.make(
            'shop.PurchaseHistory',
            customer__name='Bob'
        )
```

5.1.5 Creating Files

Model Bakery does not create files for FileField types. If you need to have the files created, you can pass the flag `_create_files=True` (defaults to `False`) to either `baker.make` or `baker.make_recipe`.

Important: the lib does not do any kind of file clean up, so it's up to you to delete the files created by it.

5.1.6 Non persistent objects

If you don't need a persisted object, Model Bakery can handle this for you as well with the **prepare** method:

```
from model_bakery import baker

customer = baker.prepare('shop.Customer')
```

It works like `make` method, but it doesn't persist the instance neither the related instances.

If you want to persist only the related instances but not your model, you can use the `_save_related` parameter for it:

```
from model_bakery import baker

history = baker.prepare('shop.PurchaseHistory', _save_related=True)
assert history.id is None
assert bool(history.customer.id) is True
```

5.1.7 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_bakery import baker

customers = baker.make('shop.Customer', _quantity=3)
assert len(customers) == 3
```

It also works with `prepare`:

```
from model_bakery import baker

customers = baker.prepare('shop.Customer', _quantity=3)
assert len(customers) == 3
```

5.2 Recipes

If you're not comfortable with random data or even if 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.

It's also possible to store the Recipes in a module called *baker_recipes.py* at your app's root directory. This recipes can later be used with the `make_recipe` function:

```
shop/
  migrations/
  __init__.py
  admin.py
  apps.py
  baker_recipes.py   <--- where you should place your Recipes
  models.py
  tests.py
  views.py
```

File: **baker_recipes.py**

```
from model_bakery.recipe import Recipe
from shop.models import Customer

customer_joe = Recipe(
    Customer,
    name='John Doe',
    nickname='joe',
    age=18,
    birthday=date.today(),
    last_shopping=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**

```
from django.test import TestCase

from model_bakery import baker

from shop.models import Customer, Contact

class CustomerTestModel(TestCase):

    def setUp(self):
        # Load the recipe 'customer_joe' from 'shop/baker_recipes.py'
        self.customer_one = baker.make_recipe(
            'shop.customer_joe'
        )
```

Or if you don't want a persisted instance:

```
from model_bakery import baker

baker.prepare_recipe('shop.customer_joe')
```

Another examples

Note: You can use the _quantity parameter as well if you want to create more than one object from a single recipe.

You can define recipes locally to your module or test case as well. This can be useful for cases where a particular set of values may be unique to a particular test case, but used repeatedly there. For example:

File: **baker_recipes.py**

```
company_recipe = Recipe(Company, name='WidgetCo')
```

File: **test_model.py**

```
class EmployeeTest(TestCase):
    def setUp(self):
        self.employee_recipe = Recipe(
            Employee,
            name=seq('Employee '),
            company=baker.make_recipe('app.company_recipe')
        )

    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....
```

5.2.1 Recipes with foreign keys

You can define `foreign_key` relations:

```
from model_bakery.recipe import Recipe, foreign_key
from shop.models import Customer, PurchaseHistory

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

history = Recipe(PurchaseHistory,
    customer=foreign_key(customer)
)
```

Notice that `customer` is a *recipe*.

You may be thinking: "I can put the Customer 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_bakery.recipe import related, Recipe
from shop.models import Customer, PurchaseHistory

history = Recipe(PurchaseHistory)
customer_with_2_histories = Recipe(Customer,
    name='Albert',
    purchasehistory_set=related('history', 'history'),
)
```

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:

```
from model_bakery.recipe import related, Recipe

pencil = Recipe(Product, name='Pencil')
pen = Recipe(Product, name='Pen')
history = Recipe(PurchaseHistory)

history_with_prods = history.extend(
    products=related(pencil, pen)
)
```

5.2.2 Recipes with callables

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

```
from datetime import date
from model_bakery.recipe import Recipe
from shop.models import Customer
```

(continues on next page)

(continued from previous page)

```
customer = Recipe(
    Customer,
    birthday=date.today,
)
```

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

5.2.3 Recipes with iterators

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

```
from itertools import cycle

names = ['Ada Lovelace', 'Grace Hopper', 'Ida Rhodes', 'Barbara Liskov']
customer = Recipe(Customer,
    name=cycle(names)
)
```

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

5.2.4 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_bakery.recipe import Recipe, seq
from shop.models import Customer

customer = Recipe(Customer,
    name=seq('Joe'),
    age=seq(15)
)

customer = baker.make_recipe('shop.customer')
customer.name
>>> 'Joe1'
customer.age
>>> 16

new_customer = baker.make_recipe('shop.customer')
new_customer.name
>>> 'Joe2'
new_customer.age
>>> 17
```

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

Sequences and iterables can be used not only for recipes, but with `baker.make` as well:

```
# it can be imported directly from model_bakery
from model_bakery import seq
from model_bakery import baker
```

(continues on next page)

(continued from previous page)

```
customer = baker.make('Customer', name=seq('Joe'))
customer.name
>>> 'Joe1'

customers = baker.make('Customer', name=seq('Chad'), _quantity=3)
for customer in customers:
    print(customer.name)
>>> 'Chad1'
>>> 'Chad2'
>>> 'Chad3'
```

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 date, timedelta
from model_bakery.recipe import Recipe, seq
from shop.models import Customer


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

customer = baker.make_recipe('shop.customer')
customer.age
>>> 18
customer.height_ft
>>> 5.75
customer.appointment
>>> datetime.date(2014, 7, 22)

new_customer = baker.make_recipe('shop.customer')
new_customer.age
>>> 21
new_customer.height_ft
>>> 6.0
new_customer.appointment
>>> datetime.date(2014, 7, 23)
```

5.2.5 Overriding recipe definitions

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

```
from model_bakery import baker

baker.make_recipe('shop.customer', name='Ada Lovelace')
```

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

5.2.6 Recipe inheritance

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

```
customer = Recipe(
    Customer,
    bio='Some customer bio',
    age=30,
    happy=True,
)
sad_customer = customer.extend(
    happy=False,
)
```

5.3 How Model Bakery behaves?

By default, Model Bakery 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 model_bakery import baker

customer = baker.make('shop.Customer', happy=True, bio='Happy customer')
```

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

```
customer = baker.make('shop.Customer', _fill_optional=['happy', 'bio'])
```

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

```
customer = baker.make('shop.Customer', _fill_optional=True)
```

5.3.1 When shouldn't you let Baker generate things for you?

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

Model Bakery 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.

5.3.2 Currently supported fields

- `BooleanField`, `NullBooleanField`, `IntegerField`, `BigIntegerField`, `SmallIntegerField`, `PositiveIntegerField`, `PositiveSmallIntegerField`, `FloatField`, `DecimalField`

- `CharField`, `TextField`, `BinaryField`, `SlugField`, `URLField`, `EmailField`, `IPAddressField`, `GenericIPAddressField`, `ContentType`
- `ForeignKey`, `OneToOneField`, `ManyToManyField` (even with through model)
- `DateField`, `DateTimeField`, `TimeField`, `DurationField`
- `FileField`, `ImageField`
- `JSONField`, `ArrayField`, `HStoreField`
- `CICharField`, `CIEmailField`, `CITextField`

Require `django.contrib.gis` in `INSTALLED_APPS`:

- `GeometryField`, `PointField`, `LineStringField`, `PolygonField`, `MultiPointField`, `MultiLineStringField`, `MultiPolygonField`, `GeometryCollectionField`

5.3.3 Custom fields

Model Bakery allows you to define generators methods for your custom fields or overrides its default generators. This can be achieved by specifing the field and generator function for the `generators.add` function. Both can be the real python objects imported in settings or just specified as import path string.

Examples:

```
from model_bakery import baker

def gen_func():
    return 'value'

baker.generators.add('test.generic.fields.CustomField', gen_func)
```

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

# in your tests.py file:
from model_bakery import baker

baker.generators.add('test.generic.fields.CustomField', 'code.path.gen_func')
```

5.3.4 Customizing Baker

In some rare cases, you might need to customize the way Baker base class behaves. This can be achieved by creating a new class and specifying it in your settings files. It is likely that you will want to extend Baker, however the minimum requirement is that the custom class have `make` and `prepare` functions. In order for the custom class to be used, make sure to use the `model_bakery.baker.make` and `model_bakery.baker.prepare` functions, and not `model_bakery.baker.Baker` directly.

Examples:

```
# in the module code.path:
class CustomBaker(baker.Baker)
    def get_fields(self):
        return [
            field
```

(continues on next page)

(continued from previous page)

```
            for field in super(CustomBaker, self).get_fields()
            if not field isinstance CustomField
        ]

# in your settings.py file:
BAKER_CUSTOM_CLASS = 'code.path.CustomBaker'
```

Additionaly, if you want to your created instance to be returned respecting one of your custom ModelManagers, you can use the `_from_manager` parameter as the example bellow:

```
movie = baker.make(Movie, title='Old Boys', _from_manager='availables')  # This will␣
↪use the Movie.availables model manager
```

5.3.5 Save method custom parameters

If you have overwritten the `save` method for a model, you can pass custom parameters to it using Model Bakery. Example:

```
class ProjectWithCustomSave(models.Model)
    # some model fields
    created_by = models.ForeignKey(settings.AUTH_USER_MODEL)

    def save(self, user, *args, **kwargs):
        self.created_by = user
        return super(ProjectWithCustomSave, self).save(*args, **kwargs)

#with model baker:
user = baker.make(settings.AUTH_USER_MODEL)
project = baker.make(ProjectWithCustomSave, _save_kwargs={'user': user})
assert user == project.user
```

5.4 Migrating from Model Mommy

Model Bakery has a Python script to help you to migrate your project's test code from Model Mommy to Model Bakery. This script will rename recipe files and replace legacy imports by the new ones.

From your project's root dir, execute the following commands:

```
$ pip uninstall model_mommy
$ pip install model_bakery
$ wget https://raw.githubusercontent.com/model-bakers/model_bakery/master/utils/from_
↪mommy_to_bakery.py
$ python from_mommy_to_bakery.py --dry-run  # will list the files that'll be changed
$ python from_mommy_to_bakery.py            # migrate from model_mommy to model_bakery
$ python manage.py test
```

This command will only migrate `*.py` files. Any other file type such as `tox.ini`, `requirements.txt` etc, have to be updated manually.