

---

# Mixer Documentation

*Release 6.0.1*

**Kirill Klenov**

**Mar 30, 2018**



---

## Contents

---

<b>1</b>	<b>User's Guide</b>	<b>3</b>
1.1	Installation . . . . .	3
1.2	Quickstart . . . . .	4
<b>2</b>	<b>API Reference</b>	<b>9</b>
2.1	API . . . . .	9
<b>3</b>	<b>Bug tracker</b>	<b>17</b>
	<b>Python Module Index</b>	<b>19</b>





Welcome to Mixer's documentation. Mixer is an object generation tool for your application.

It's supported [Django ORM](#), [SQLAlchemy ORM](#), [Pony ORM](#), [Peewee ORM](#), [Mongoengine ODM](#) and etc.

Mixer is very useful for testing and fixtures replacement.

**copyright** 2013 by Kirill Klenov.

**license** BSD, see LICENSE for more details.



This part of the documentation will show you how to get started in using Mixer.

## 1.1 Installation

### Contents

- *Installation*

- 
- Django (1.10, 1.11, 2.0) for Django ORM support;
  - Faker  $\geq 0.7.3$
  - Flask-SQLAlchemy for SQLAlchemy ORM support and integration as Flask application;
  - Mongoengine for Mongoengine ODM support;
  - SQLAlchemy for SQLAlchemy ORM support;
  - fake-factory  $\geq 0.5.0$
  - faker  $== 0.7.3$
  - python 2.7 or 3.6+
- 

**Mixer** should be installed using pip:

```
pip install mixer
```

## 1.2 Quickstart

### Contents

- *Quickstart*
  - *Django ORM*
    - \* *Models*
    - \* *Base Usage*
    - \* *Blend models with values*
  - *SQLAlchemy ORM*
    - \* *Support for Flask-SQLAlchemy models that have \_\_init\_\_ arguments*
  - *Flask integration*
  - *Mongoengine*

Mixer is easy to use and really fun for testing applications. Module has a common api for all backends (Django, Flask).

### 1.2.1 Django ORM

#### Models

Somewhere in 'someapp/models.py':

```
from django.db import models

class Client(models.Model):
    username = models.CharField(max_length=20)
    name = models.CharField(max_length=50)
    created_at = models.DateField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    score = models.IntegerField(default=50)

class Message(models.Model):
    content = models.TextField()
    client = models.ForeignKey(Client)

class Tag(models.Model):
    title = models.CharField(max_length=20)
    messages = models.ManyToManyField(Message, null=True, blank=True)
```

#### Base Usage

You can use class or string with model name.

```
from mixer.backend.django import mixer

# Generate model's instance and save to db
message = mixer.blend('someapp.message')
```



```

print message.content # Some like --> necessitatibus voluptates animi molestiae_
↳dolores...

print message.client.username # Some like --> daddy102

print message.client.name # Some like --> Clark Llandrindod

# Generate a few pieces
messages = mixer.cycle(4).blend('someapp.message')

```

## Blend models with values

```

from mixer.backend.django import mixer

# Generate model with some values
client = mixer.blend(Client, username='test')
assert client.username == 'test'

# Generate model with reference
message = mixer.blend(Message, client__username='test2')
assert message.client.username == 'test2'

# Value may be callable
client = mixer.blend(Client, username=lambda:'callable_value')
assert client.username == 'callable_value'

# Value may be a generator
clients = mixer.cycle(4).blend(Client, username=(name for name in ('Piter', 'John')))

# Value could be getting a counter
clients = mixer.cycle(4).blend(Client, username=mixer.sequence(lambda c: "test_%s" %
↳c))
print clients[2].username # --> 'test_2'

# Short format for string formatting
clients = mixer.cycle(4).blend(Client, username=mixer.sequence("test_{0}"))
print clients[2].username # --> 'test_2'

# Force to generation of a default (or null) values
client = mixer.blend(Client, score=mixer.RANDOM)
print client.score # Some like: --> 456

# Set related values from db by random
message = mixer.blend(Message, client=mixer.SELECT)
assert message.client in Client.objects.all()

```

## 1.2.2 SQLAlchemy ORM

```

from mixer.backend.sqlalchemy import mixer

# Generate model's instance and save to db
message = mixer.blend('path.to.module.ModelClass')

print message.content # Some like --> necessitatibus voluptates animi molestiae_
↳dolores...

```

```
print message.client.username # Some like --> daddy102

print message.client.name # Some like --> Clark Llandrindod

# Generate a few pieces
messages = mixer.cycle(4).blend('path.to.module.ModelClass')
```

### Support for Flask-SQLAlchemy models that have `__init__` arguments

To support this scheme, just create your own mixer class, like this:

```
from mixer.backend.sqlalchemy import Mixer

class MyOwnMixer(Mixer):

    def populate_target(self, values):
        target = self.__scheme(**values)
        return target

mixer = MyOwnMixer()
```

### 1.2.3 Flask integration

Module integrate the Mixer to Flask application.

See example:

```
from mixer.backend.flask import mixer

mixer.init_app(flask_app)

user = mixer.blend('path.to.models.User')
```

### 1.2.4 Mongoengine

Support for Mongoengine ODM.

---

**Note:** Support for [Mongoengine](#) is in early development.

---

```
from mixer.backend.mongoengine import mixer

class User(Document):
    created_at = DateTimeField(default=datetime.datetime.now)
    email = EmailField(required=True)
    first_name = StringField(max_length=50)
    last_name = StringField(max_length=50)

class Post(Document):
    title = StringField(max_length=120, required=True)
    author = ReferenceField(User)
    tags = ListField(StringField(max_length=30))
```

```
post = mixer.blend(Post, author__username='foo')
```



If you are looking for information on a specific function, class or method, this part of the documentation is for you.

## 2.1 API

### Contents

- *API*
  - *Common interface*
  - *Set values*
  - *Force a random values*
  - *Force a fake values*
  - *Select a relation from database*
  - *Virtual mixed object*

This part of the documentation documents all the public classes and functions in Mixer.

### 2.1.1 Common interface

```
class mixer.main.Mixer (fake=True, factory=None, loglevel=30, silence=False, locale='en_US',  
                        **params)
```

This class is using for integration to an application.

#### Parameters

- **fake** – (True) Generate fake data instead of random data.
- **factory** – (*GenFactory*) Fabric's factory

```
class SomeScheme:
    score = int
    name = str

mixer = Mixer()
instance = mixer.blend(SomeScheme)
print instance.name # Some like: 'Mike Douglass'

mixer = Mixer(fake=False)
instance = mixer.blend(SomeScheme)
print instance.name # Some like: 'AKJfdjh3'
```

**type\_mixer\_cls**

alias of TypeMixer

**blend** (*scheme*, *\*\*values*)Generate instance of *scheme*.**Parameters**

- **scheme** – Scheme class for generation or string with class path.
- **values** – Keyword params with predefined values

**Return value** A generated instance

```
mixer = Mixer()

mixer.blend(SomeScheme, active=True)
print scheme.active # True

mixer.blend('module.SomeScheme', active=True)
print scheme.active # True
```

**get\_typemixer** (*scheme*)

Return a cached typemixer instance.

**Return TypeMixer****static postprocess** (*target*)

Run the code after generation.

**Return target****static sequence** (*\*args*)

Create a sequence for predefined values.

It makes a infinity loop with given function where does increment the counter on each iteration.

**Parameters args** – If method get more one arguments, them make generator from arguments (loop on arguments). If that get one argument and this equal a function, method makes a generator from them. If argument is equal string it should be using as format string.

By default function is equal 'lambda x: x'.

**Returns** A generator

Mixer can uses a generators.

```
gen = (name for name in ['test0', 'test1', 'test2'])
for counter in range(3):
    mixer.blend(Scheme, name=gen)
```

Mixer.sequence is a helper for create generators more easy.

Generate values from sequence:

```
for _ in range(3):
    mixer.blend(Scheme, name=mixer.sequence('john', 'mike'))
```

Make a generator from function:

```
for counter in range(3):
    mixer.blend(Scheme, name=mixer.sequence(
        lambda c: 'test%s' % c
    ))
```

Short format is a python formatting string

```
for counter in range(3):
    mixer.blend(Scheme, name=mixer.sequence('test{0}'))
```

**cycle** (*count=5*)

Generate a few objects. The syntastic sugar for cycles.

**Parameters** *count* – List of objects or integer.

**Returns** ProxyMixer

```
users = mixer.cycle(5).blend('somemodule.User')

profiles = mixer.cycle(5).blend(
    'somemodule.Profile', user=(user for user in users)

apples = mixer.cycle(10).blend(
    Apple, title=mixer.sequence('apple_{0}'))
```

**middleware** (*scheme*)

Middleware decorator.

You could add the middleware layers to generation process:

```
from mixer.backend.django import mixer

# Register middleware to model
@mixer.middleware('auth.user')
def encrypt_password(user):
    user.set_password('test')
    return user
```

You can add several middlewares. Each middleware should get one argument (generated value) and return them.

**register** (*scheme, \*\*params*)

Manually register a function as value's generator for class.field.

**Parameters**

- **scheme** – Scheme for generation (class or class path)
- **params** – Kwargs with generator's definitions (field\_name=field\_generator)

```
class Scheme:
    id = str
```

```
    title = str

def func():
    return 'ID'

mixer.register(
    Scheme,
    id=func,
    title='Always same',
)

test = mixer.blend(Scheme)
test.id == 'ID'
test.title == 'Always same'
```

**ctx** (\*args, \*\*kws)  
Redefine params for current mixer as context.

```
with mixer.ctx(commit=False):
    hole = mixer.blend(Hole)
    self.assertTrue(hole)
    self.assertFalse(Hole.objects.count())
```

**reload** (\*objs)  
Reload the objects from storage.

**guard** (\*args, \*\*kwargs)  
Abstract method. In some backends used for prevent object creation.

**Returns** A Proxy to mixer

**class** mixer.main.**GenFactory**  
Make generators for types.

**classmethod** **cls\_to\_simple** (fcls)  
Translate class to one of simple base types.

**Return type** A simple type for generation

**static** **name\_to\_simple** (fname)  
Translate name to one of simple base names.

**Return** str

**classmethod** **get\_fabric** (fcls, fname=None, fake=False)  
Make a objects fabric based on class and name.

**Return** function

## 2.1.2 Set values

**class** mixer.mix\_types.**Field** (scheme, name, \*\*params)  
Set field values.

By default the mixer generates random or fake a field values by types of them. But you can set some values by manual.

```
# Generate a User model
mixer.blend(User)
```



```
# Generate with some values
mixer.blend(User, name='John Connor')
```

---

**Note:** Value may be a callable or instance of generator.

---

```
# Value may be callable
client = mixer.blend(Client, username=lambda:'callable_value')
assert client.username == 'callable_value'

# Value may be a generator
clients = mixer.cycle(4).blend(
    Client, username=(name for name in ('Piter', 'John')))
```

**See also:**

`mixer.main.Fake`, `mixer.main.Random`, `mixer.main.Select`, `mixer.main.Mixer.sequence()`

### 2.1.3 Force a random values

**class** `mixer.mix_types.Random` (*scheme=None, \*choices, \*\*params*)

Force a *random* value.

If you initialized a *Mixer* by default mixer try to fill fields with *fake* data. You can use `mixer.RANDOM` for prevent this behaviour for a custom fields.

```
mixer = Mixer()
user = mixer.blend(User)
print user.name # Some like: Bob Marley

user = mixer.blend(User, name=mixer.RANDOM)
print user.name # Some like: Fdjw4das
```

You can setup a field type for generation of fake value:

```
user = mixer.blend(User, score=mixer.RANDOM(str))
print user.score # Some like: Fdjw4das
```

Or you can get random value from choices:

```
user = mixer.blend(User, name=mixer.RANDOM('john', 'mike'))
print user.name # mike or john
```

---

**Note:** This is also useful on ORM model generation for randomize fields with default values (or null).

---

```
from mixer.backend.django import mixer

mixer.blend('auth.User', first_name=mixer.RANDOM)
print user.first_name # Some like: Fdjw4das
```

## 2.1.4 Force a fake values

**class** `mixer.mix_types.Fake` (*scheme=None, \*choices, \*\*params*)  
Force a *fake* value.

If you initialized a *Mixer* with *fake=False* you can force a *fake* value for field with this attribute (`mixer.FAKE`).

```
mixer = Mixer(fake=False)
user = mixer.blend(User)
print user.name # Some like: Fdjw4das

user = mixer.blend(User, name=mixer.FAKE)
print user.name # Some like: Bob Marley
```

You can setup a field type for generation of fake value:

```
user = mixer.blend(User, score=mixer.FAKE(str))
print user.score # Some like: Bob Marley
```

---

**Note:** This is also usefull on ORM model generation for filling a fields with default values (or null).

---

```
from mixer.backend.django import mixer

user = mixer.blend('auth.User', first_name=mixer.FAKE)
print user.first_name # Some like: John
```

## 2.1.5 Select a relation from database

**class** `mixer.mix_types.Select` (*scheme=None, \*choices, \*\*params*)  
Select values from database.

When you generate some ORM models you can set value for related fields from database (select by random).

Example for Django (select user from exists):

```
from mixer.backend.django import mixer

mixer.generate(Role, user=mixer.SELECT)
```

You can setup a Django or SQLAlchemy filters with *mixer.SELECT*:

```
from mixer.backend.django import mixer

mixer.generate(Role, user=mixer.SELECT(
    username='test'
))
```

## 2.1.6 Virtual mixed object

**class** `mixer.mix_types.Mix` (*value=None, parent=None*)  
Virtual link on the mixed object.

```
mixer = Mixer()

# here `mixer.MIX` points on a generated `User` instance
user = mixer.blend(User, username=mixer.MIX.first_name)

# here `mixer.MIX` points on a generated `Message.author` instance
message = mixer.blend(Message, author__name=mixer.MIX.login)

# Mixer mix can get a function
message = mixer.blend(Message, title=mixer.MIX.author(
    lambda author: 'Author: %s' % author.name
))
```



## CHAPTER 3

---

### Bug tracker

---

If you have any suggestions, bug reports or annoyances please report them to the issue tracker at <https://github.com/klen/mixer/issues>

---

Development of mixer happens at Github: <https://github.com/klen/mixer>



**m**

`mixer`, 1

`mixer.backend.flask`, 6

`mixer.backend.mongoengine`, 6





## B

blend() (mixer.main.Mixer method), 10

## C

cls\_to\_simple() (mixer.main.GenFactory class method), 12

ctx() (mixer.main.Mixer method), 12

cycle() (mixer.main.Mixer method), 11

## F

Fake (class in mixer.mix\_types), 14

Field (class in mixer.mix\_types), 12

## G

GenFactory (class in mixer.main), 12

get\_fabric() (mixer.main.GenFactory class method), 12

get\_typemixer() (mixer.main.Mixer method), 10

guard() (mixer.main.Mixer method), 12

## M

middleware() (mixer.main.Mixer method), 11

Mix (class in mixer.mix\_types), 14

Mixer (class in mixer.main), 9

mixer (module), 1

mixer.backend.flask (module), 6

mixer.backend.mongoengine (module), 6

## N

name\_to\_simple() (mixer.main.GenFactory static method), 12

## P

postprocess() (mixer.main.Mixer static method), 10

## R

Random (class in mixer.mix\_types), 13

register() (mixer.main.Mixer method), 11

reload() (mixer.main.Mixer method), 12

## S

Select (class in mixer.mix\_types), 14

sequence() (mixer.main.Mixer static method), 10

## T

type\_mixer\_cls (mixer.main.Mixer attribute), 10