DDF Documentation

Release 1.7.0

Paulo Cheque

Aug 31, 2017

Contents

Django Dynamic Fixture (DDF) is a complete and simple library to create dynamic model instances for testing purposes.

It lets you focus on your tests, instead of focusing on generating some dummy data which is boring and polutes the test source code.

CHAPTER 1

Getting Started

Basic Example of Usage

Models sample:

```
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=255)

class Book(models.Model):
    name = models.CharField(max_length=255)
    authors = models.ManyToManyField(Author)
```

Test sample:

```
from django.test import TestCase
from django_dynamic_fixture import G

class SearchingBooks(TestCase):
    def test_search_book_by_author(self):
        author1 = G(Author)
        author2 = G(Author)
        book1 = G(Book, authors=[author1])
        book2 = G(Book, authors=[author2])
        books = Book.objects.search_by_author(author1.name)
        self.assertTrue(book1 in books)
        self.assertTrue(book2 not in books)
```

Installation

```
pip install django-dynamic-fixture
```

or:

```
1. Download zip file
2. Extract it
3. Execute in the extracted directory: python setup.py install
```

Development version

```
pip install -e git+git@github.com:paulocheque/django-dynamic-fixture.git#egg=django-
↪dynamic-fixture
```

requirements.txt

```
django-dynamic-fixture==<VERSION>
# or use the development version
git+git://github.com/paulocheque/django-dynamic-fixture.git#egg=django-dynamic-fixture
```

Upgrade

```
pip install django-dynamic-fixture --upgrade --no-deps
```

Compatibility

- Legacy: Django 1.2, 1.3 - Python: 2.7
- Django 1.4 with Python 2.7
- Django 1.5 with Python 2.7
- Django 1.6 with Python 2.7 or 3.3
- Not tested with Django 1.7 yet

List of features

- Highly customizable: you can customize fields recursively
- Deals with unique=True
- Deals with cyclic dependencies (including self references)
- Deals with many to many relationship (common M2M or M2M with additional data, i.e. through='table')
- Deals with custom fields (especially if the custom field inherits from a django field)
- Support for parallel tests

- Deals with auto calculated attributes
- It is easy to debug errors

Motivation

- It is a terrible practice to use static data in tests.
- Creating dynamic fixture for each model is boring and it produces a lot of replicated code.
- It is a bad idea to use uncontrolled data in tests, like bizarre random data.

Comparison with other tools

The DDF was created in a context of a project with a very complex design with many cyclic dependencies. In that context, no available tool was satisfactory. Or we stumbled in some infinite loop or some bug caused by a custom field. For these reasons, the tests started to fail and it was very hard to understand why.

Another thing, the DDF was planned to have a lean and clean syntax. We believe that automated tests must be developed quickly with the minimum overhead. For that reason we are in favor of less verbose approach, except in the documentation ;)

Also, DDF is flexible, since it is possible to customize the entire data generation or by field.

- Either they are incomplete, or bugged or it produces erratic tests, because they use random and uncontrolled data.
- The syntax of others tools is too verbose, which polutes the tests.
- Complete, lean and practice documentation.
- It is hard to debug tests with another tools.
- List of other tools: <https://www.djangopackages.com/grids/g/testing/> or <http://djangopackages.com/grids/g/fixtures>
- The core of the tool is the algorithm, it is not the data generation as all other tools. This means you can change the data generation logic.

Plus:

- Nose plugin that enables a setup for the entire suite (unittest2 includes only setups for class and module)
- Nose plugin to count how many queries are executed by test
- Command to count how many queries are executed to save any kind of model instance
- FileSystemDjangoTestCase that facilitates to create tests for features that use filesystem.

External references

- http://stackoverflow.com/search?q=django+dynamic+fixture
- http://stackoverflow.com/questions/12487337/optimizing-setup-and-teardown-for-sample-django-model-using-django-nose-and
- http://stackoverflow.com/questions/4400609/initial-data-fixture-management-in-django

CHAPTER 2

Main features

Get: G

The **G** function is the main feature of DDF. It receives a model class and it will return a valid and persisted instance filled with dynamically generated data:

```
from django_dynamic_fixture import G, get
instance = G(MyModel)
# The same as:
# instance = get(MyModel)
print instance.id # this will print the ID, indicating the instance was saved
print instance.some_field # this will print the auto generated data
```

This facilitates writing tests and it hides all dummy data that polutes the source code. But all important data of the test must be explicitily defined:

```
instance = G(MyModel, my_field=123, another_field='abc')
print instance.my_field # this will print 123
print instance.another_field # this will print 'abc'
```

Important details:

- The id (AutoField) is auto filled, unless you set a value to it.
- if a field has default value, it is used by default.
- if a field has choices, first option is selected by default.

New: N

This function **N** is similar to **G**, except it will not save the generated instance, only all internal dependencies of *ForeignKey* and *OneToOneField* fields. Since the instance does not have an ID, it can not insert instances in *ManyToManyField* fields. This function may be useful mainly in one of the following situations: create a unit test independent of

the database; or create a not persisted instance that will be manipulated before saving it (usually dealing with custom fields, custom validations etc):

```
from django_dynamic_fixture import N, new
instance = N(MyModel)
# The same as:
# instance = new(MyModel)
print instance.id # this will print None
print instance.some_field # this will print the auto generated data
```

It is possible to disable saving all instances, but it has to be disabled manually:

```
instance = N(MyModel, persist_dependencies=False)
print instance.id # this will print None
print instance.some_fk_field.id # this will print None
```

Fixture: F

It is possible to explicitly set a value for a relationship field(*ForeingKey*, *OneToOneField* and *ManyToManyField*) using the **F** function:

```
from django_dynamic_fixture import G, F, fixture
instance = G(MyModel, my_fk_field=F(my_field=1000))
# The same as:
# instance = G(MyModel, my_fk_field=fixture(my_field=1000))
print instance.my_fk_field.my_field # this will print 1000
```

This is the equivalent to:

```
my_fk_field = G(MyOtherModel, my_field=1000)
instance = G(MyModel, my_fk_field=my_fk_field)
print instance.my_fk_field.my_field # this will print 1000
```

F function is recursible:

```
instance = G(MyModel, fk_a=F(fk_b=F(field_c=1000)))
print instance.fk_a.fk_b.field_c # this will print 1000
```

F can be used to customize instances of a ManyToManyField too:

```
instance = G(MyModel, many_to_many_field=[F(field_x=1), F(field_x=2)])
print instance.many_to_many_field.all()[0].field_x # this will print 1
print instance.many_to_many_field.all()[1].field_x # this will print 2
```

Many to Many fields

DDF can add instances in *ManyToManyFields*, but only if the instance has been persisted to the database (Django requirement). DDF handles *ManyToManyFields* with *through* table as well.

It is possible to define how many instances will be created dynamically:

```
instance = G(MyModel, many_to_many_field=5)
print instance.many_to_many_field.all().count() # this will print 5
```

It is possible to customize each instance of the *ManyToManyField*:

```
instance = G(MyModel, many_to_many_field=[F(field_x=10), F(field_y=20)])
print instance.many_to_many_field.all().count() # this will print 2
print instance.many_to_many_field.all()[0].field_x # this will print 10
print instance.many_to_many_field.all()[1].field_y # this will print 20
```

It is possible to pass already created instances too:

```
instance1 = G(MyRelatedModel)
instance2 = G(MyRelatedModel)

instance = G(MyModel, many_to_many_field=[instance1, instance2])
print instance.many_to_many_field.all().count() # this will print 2

instance = G(MyModel, many_to_many_field=[F(), instance1, F(), instance2])
print instance.many_to_many_field.all().count() # this will print 4
```

Django Look Up fields syntax (New in 1.6.1)

This is an alias to F function, but it follows the Django pattern of filters that use two underlines to access internal fields of foreign keys:

```
from django_dynamic_fixture import G
instance = G(MyModel, myfkfield__myfield=1000)
print instance.myfkfield__myfield # this will print 1000
```

Just be careful because DDF does not interpret related names yet.

Global Settings

You can configure DDF in `settings.py` file. You can also override the global config per instance creation when necessary.

- **DDF_FILL_NULLABLE_FIELDS** (Default = True): DDF can fill nullable fields (`null=True`) with None or some data:

```
# SomeModel(models.Model): nullable_field = Field(null=True)
G(SomeModel).nullable_field is None # True if DDF_FILL_NULLABLE_FIELDS is True
G(SomeModel).nullable_field is None # False if DDF_FILL_NULLABLE_FIELDS is False

# You can override the global config for one case:
G(Model, fill_nullable_fields=False)
G(Model, fill_nullable_fields=True)
```

- **DDF_VALIDATE_MODELS** (Default = False): DDF will call `model.full_clean()` method before saving to the database:

```
# You can override the global config for one case:
G(Model, validate_models=True)
G(Model, validate_models=False)
```

- **DDF_VALIDATE_ARGS** (Default = False): DDF can raise an exception if you are trying to fill data for a non existent field:

```
G(Model, nonexistent_field='data') # DDF will ignore it if DDF_VALIDATE_ARGS is
↪False
G(Model, nonexistent_field='data') # DDF will raise an exception if DDF_VALIDATE_
↪ARGS is True

# You can override the global config for one case:
G(Model, validate_args=False)
G(Model, validate_args=True)
```

- **DDF_FIELD_FIXTURES** (Default = {}) (new in 1.8.0): Dictionary where the key is the full qualified name of the field and the value is a function without parameters that returns a value:

```
DDF_FIELD_FIXTURES = {'path.to.your.Field': lambda: random.randint(0, 10) }
```

- **DDF_USE_LIBRARY** (Default = False): For using the Shelve feature:

```
# You can override the global config for one case:
G(Model, use_library=False)
G(Model, use_library=True)
```

- **DDF_NUMBER_OF_LAPS** (Default = 1): For models with foreign keys to itself (`ForeignKey('self')`), DDF will avoid infinite loops because it stops creating objects after it create **n laps** for the cycle:

```
# You can override the global config for one case:
G(Model, number_of_laps=5)
```

- **DDF_DEBUG_MODE** (Default = False): To show some DDF logs:

```
# You can override the global config for one case:
G(Model, debug_mode=True)
G(Model, debug_mode=False)
```

CHAPTER 3

Data Fixtures

This configuration defines the algorithm of data generation that will be used to populate fields with dynamic data. Do NOT mix data fixtures in the same test suite because the generated data may conflict and it will produce erratic tests.

In settings.py:

```
DDF_DEFAULT_DATA_FIXTURE = 'sequential' # or 'static_sequential' or 'random' or 'path.
↪to.your.DataFixtureClass'
```

Overriding global data fixture

This algorithm will be used just for the current model generation.

In the test file or shell:

```
G(MyModel, data_fixture='sequential')
G(MyModel, data_fixture='random')
G(MyModel, data_fixture=MyCustomDataFixture())
```

Sequential Data Fixture

Useful to use in test suites. Sequential Data Fixture stores an independent counter for each model field that is incremented every time this field has to be populated. If for some reason the field has some restriction (*max_length*, *max_digits* etc), the counter restarts. This counter is used to populate fields of numbers (*Integer*, *BigDecimal* etc) and strings (*CharField*, *TextField* etc). For *BooleanFields*, it will always return False. For *NullBooleanFields*, it will always return None. For date and time fields, it will always return Today minus 'counter' days or Now minus 'counter' seconds, respectively.

In settings.py:

```
DDF_DEFAULT_DATA_FIXTURE = 'sequential'
```

In the test file:

```
instance = G(MyModel)
print instance.integerfield_a # this will print 1
print instance.integerfield_b # this will print 1
print instance.charfield_b # this will print 1
print instance.booleanfield # this will print False
print instance.nullbooleanfield # this will print None

instance = G(MyModel)
print instance.integerfield_a # this will print 2
print instance.integerfield_b # this will print 2
print instance.charfield_b # this will print 2
print instance.booleanfield # this will print False
print instance.nullbooleanfield # this will print None

instance = G(MyOtherModel)
print instance.integerfield_a # this will print 1

# ...
```

Static Sequential Data Fixture

Useful to use in test suites. Static Sequential Data Fixture is the same as Sequential Data Fixture, except it will increase the counter only if the field has *unique=True*.

In settings.py:

```
DDF_DEFAULT_DATA_FIXTURE = 'static_sequential'
```

In the test file:

```
instance = G(MyModel)
print instance.integerfield_unique # this will print 1
print instance.integerfield_notunique # this will print 1

instance = G(MyModel)
print instance.integerfield_unique # this will print 2
print instance.integerfield_notunique # this will print 1
# ...
```

Random Data Fixture

Useful to use in python shells. In shell you may want to do some manual tests, and DDF may help you to generate models too. If you are using shell with a not-in-memory database, you may have problems with *SequentialDataFixture* because the sequence will be reset every time you close the shell, but the data already generated is persisted.

It is dangerous to use this data fixture in a test suite because it can produce erratic tests. For example, depending on the quantity of tests and *max_length* of a *CharField*, there is a high probability to generate an identical value which will result in invalid data for fields with *unique=True*.

In settings.py:

```
DDF_DEFAULT_DATA_FIXTURE = 'random'
```

In the test file:

```
instance = G(MyModel)
print instance.integerfield_a # this will print a random number
print instance.charfield_b # this will print a random string
print instance.booleanfield # this will print False or True
print instance.nullbooleanfield # this will print None, False or True
# ...
```

Custom Data Fixture

In settings.py:

```
DDF_DEFAULT_DATA_FIXTURE = 'path.to.your.DataFixtureClass'
```

In the path/to/your.py file:

```
from django_dynamic_fixture.ddf import DataFixture
class DataFixtureClass(DataFixture): # it can inherit of SequentialDataFixture,␣
↪RandomDataFixture etc.
    def integerfield_config(self, field, key): # method name must have the format:␣
↪FIELDNAME_config
        return 1000 # it will always return 1000 for all IntegerField
```

In the test file:

```
instance = G(MyModel)
print instance.integerfield_a # this will print 1000
print instance.integerfield_b # this will print 1000
# ...
```

Custom Field Fixture

You can also override a field default fixture or even create a fixture for a new field using the **DDF_FIELD_FIXTURES** settings in `settings.py`:

```
# https://github.com/bradjasper/django-jsonfield
import json
DDF_FIELD_FIXTURES = {
    'jsonfield.fields.JSONCharField': {'ddf_fixture': lambda: json.dumps({'some␣
↪random value': 'c'})},
    'jsonfield.fields.JSONField': {'ddf_fixture': lambda: json.dumps([1, 2, 3])},
}
```

CHAPTER 4

Field Data Generation

Supported Fields

- **Numbers**: IntegerField, SmallIntegerField, PositiveIntegerField, PositiveSmallIntegerField, BigIntegerField, FloatField,DecimalField
- **Strings**: CharField, TextField, SlugField, CommaSeparatedintegerField
- **Booleans**: BooleanField, NullBooleanField
- **Timestamps**: DateField, TimeField, DatetimeField
- **Utilities**: EmailField, UrlField, IPAddressField, XMLField
- **Files**: FilePathField, FileField, ImageField
- **Binary**: BinaryField

PS: Use **DDF_FIELD_FIXTURES** settings, customized data or even the field default values to deal with not supported fields.

GeoDjango Fields

After *1.8.4* version, DDF has native support for GeoDjango fields: GeometryField, PointField, LineStringField, PolygonField, MultiPointField, MultiLineStringField, MultiPolygonField, GeometryCollectionField.

For older versions of DDF, please, use the following approach:

You can use DDF_FIELD_FIXTURES to create fixtures for Geo Django fields:

```
# https://docs.djangoproject.com/en/dev/ref/contrib/gis/
from django.contrib.gis.geos import Point
DDF_FIELD_FIXTURES = {
    'django.contrib.gis.db.models.GeometryField': lambda: None,
    'django.contrib.gis.db.models.PointField': lambda: None,
    'django.contrib.gis.db.models.LineStringField': lambda: None,
```

```
    'django.contrib.gis.db.models.PolygonField': lambda: None,
    'django.contrib.gis.db.models.MultiPointField': lambda: None,
    'django.contrib.gis.db.models.MultiLineStringField': lambda: None,
    'django.contrib.gis.db.models.MultiPolygonField': lambda: None,
    'django.contrib.gis.db.models.GeometryCollectionField': lambda: None,
}
```

About Custom Fields

- Customized data is also valid for unsupported fields.
- You can also set the field fixture using the **DDF_FIELD_FIXTURES** settings. (new in 1.8.0)
- if a field is not default in Django, but it inherits from a Django field, it will be filled using its config.
- if a field is not default in Django and not related with a Django field, it will raise an *UnsupportedFieldError*.
- if it does not recognize the Field class, it will raise an *UnsupportedFieldError*.

Fill Nullable Fields

This option define if nullable fields (fields with *null=True*) will receive a generated data or not (data=None). It is possible to override the global option for an specific instance. But be careful because this option will be propagate to all internal dependencies.

In settings.py:

```
DDF_FILL_NULLABLE_FIELDS = True
```

In the test file:

```
instance = G(MyModel, fill_nullable_fields=False)
print instance.a_nullable_field # this will print None
print instance.a_required_field # this will print a generated data

instance = G(MyModel, fill_nullable_fields=True)
print instance.a_nullable_field # this will print a generated data
print instance.a_required_field # this will print a generated data
```

Ignoring Fields (New in 1.2.0)

This option defines a list of fields DDF will ignore. In other words, DDF will not fill it with dynamic data. This option can be useful for fields auto calculated by models, like [MPTT](https://github.com/django-mptt/django-mptt) models. Ignored fields are propagated ONLY to self references.

In settings.py:

```
DDF_IGNORE_FIELDS = ['field_x', 'field_y'] # default = []
```

Ignored field names can use wildcard matching with '*' and '?' characters which substitute multiple or one character respectively. Wildcards are useful for fields that should not be populated and which match a pattern, like `*_ptr` fields for [django-polymorphic](https://github.com/django-polymorphic/django-polymorphic).

In settings.py:

```
DDF_IGNORE_FIELDS = ['*_ptr'] # Ignore django-polymorphic pointer fields
```

It is not possible to override the global configuration, just extend the list. So use global option with caution:

```
instance = G(MyModel, ignore_fields=['another_field_name'])
print instance.another_field_name # this will print None
```

Number of Laps

This option is used by DDF to control cyclic dependencies (*ForeignKey* by *self*, denormalizations, bad design etc). DDF does not enter infinite loop of instances generation. This option defines how many laps DDF will create for each cyclic dependency. This option can also be used to create trees with different lengths.

In settings.py:

```
DDF_NUMBER_OF_LAPS = 1
```

In the test file:

```
instance = G(MyModel, number_of_laps=1)
print instance.self_fk.id # this will print the ID
print instance.self_fk.self_fk.id # this will print None

instance = G(MyModel, number_of_laps=2)
print instance.self_fk.id # this will print the ID
print instance.self_fk.self_fk.id  # this will print the ID
print instance.self_fk.self_fk.self_fk.id # this will print None
```

Copier (New in 1.6.0)

Copier is a feature to copy the data of a field to another one. It is necessary to avoid cycles in the copier expression. If a cycle is found, DDF will alert the programmer the expression is invalid:

```
instance = G(MyModel, some_field=C('another_field'))
print instance.some_field == instance.another_field # this will print True

instance = G(MyModel, some_field=C('another_field'), another_field=50)
print instance.some_field # this will print 50
```

It is possible to copy values of internal relationships, but only in the bottom-up way:

```
instance = G(MyModel, some_field=C('some_fk_field.another_field'))
print instance.some_field == instance.some_fk_field.another_field # this will print␣
↪True
```

Default Shelve (New in 1.6.0)

Sometimes DDF can not generate a valid and persisted instance because it contains custom fields or custom validations (field or model validation). In these cases it is possible to teach DDF how to build a valid instance. It is necessary to

create a valid configuration and shelve it in an internal and global DDF library of configurations. All future instances of that model will use the shelved configuration as base. All custom configurations will override the shelved option just for the current model instance generation. But to use the default configuration it is necessary to enable the use of the DDF library.

In settings.py:

```
DDF_USE_LIBRARY = True
```

In the test file:

```
instance = G(Model, shelve=True, field_x=99)
print instance.field_x # this will print 99

instance = G(Model, use_library=True)
print instance.field_x # this will print 99

instance = G(Model, use_library=False)
print instance.field_x # this will a dynamic generated data
```

It is possible to override shelved configuration:

```
G(Model, shelve=True, field_x=888)
instance = G(Model, use_library=True, field_x=999)
print instance.field_x # this will print 999
```

It is possible to store custom functions of data fixtures for fields too:

```
zip_code_data_fixture = lambda field: 'MN 55416'
instance = G(Model, shelve=True, zip_code=zip_code_data_fixture)

instance = G(Model, use_library=True)
print instance.zip_code # this will print 'MN 55416'
```

It is possible to store Copiers too:

```
instance = G(Model, shelve=True, x=C('y'))

instance = G(Model, use_library=True, y=5)
print instance.x # this will print 5
```

If the model is used by another applications, it is important to put the code that shelve configurations in the file *your_app.tests.ddf_setup.py* because DDF can not control the order tests will be executed, so a test of other application can be executed before the valid configuration is shelved. The file *ddf_setup.py* prevents this, because it is loaded before DDF starts to generate the instance of a particular model. It works like a "setup suite", like the DDF Setup Nose plugin.

Named Shelve (New in 1.6.0)

The named shelve works like default shelve, but it has to have a name. It is possible to store more than one configuration by model type.

In settings.py:

```
DDF_USE_LIBRARY = True
```

In the test file:

```
G(Model, shelve='some name', field_x=888)
G(Model, shelve='another name', field_x=999)

instance = G(Model, named_shelve='some name', use_library=True)
print instance.field_x # this will print 888

instance = G(Model, named_shelve='another name', use_library=True)
print instance.field_x # this will print 999

instance = G(Model, named_shelve='some name', use_library=False)
print instance.field_x # this will print a dynaimc generated data
```

If a DDF does not found the named shelve, it will raise an error:

```
G(Model, named_shelve='name not found in DDF library', use_library=True)
```

It is important to note that all named shelve will inherit the configuration from the default shelve:

```
G(Model, shelve=True, x=999)
G(Model, shelve='some name', y=888)

instance = G(Model, named_shelve='some name', use_library=True)
print instance.x # this will print 999
print instance.y # this will print 888
```

CHAPTER 5

DDF Extras

Decorators (New in 1.4.0)

Example:

```
from django_dynamic_fixture.decorators import skip_for_database, only_for_database,␣
↪SQLITE3, POSTGRES

@skip_for_database(SQLITE3)
def test_some_feature_that_use_raw_sql_not_supported_by_sqlite(self): pass

@only_for_database(POSTGRES)
def test_some_feature_that_use_raw_sql_specific_to_my_production_database(self): pass
```

It is possible to pass a custom string as argument, one that would be defined in settings.DATABASES['default']['ENGINE']:

```
@skip_for_database('my custom driver')
@only_for_database('my custom driver')
```

Validation

Validate Models (New in 1.5.0)

This option is a flag to determine if the method *full_clean* of model instances will be called before DDF calls the save method.

In settings.py:

```
DDF_VALIDATE_MODELS = False
```

In the test file:

```
G(MyModel, validate_models=True)
```

Validate Arguments (New in 1.5.0)

This option is a flag to determine if field names passed to **G** and **N** function will be validated. In other words, DDF will check if the model has the field.

In settings.py:

```
DDF_VALIDATE_ARGS = True # default = False for compatiblity reasons, but it is␣
↪recommended to activate this option.
```

In the test file:

```
G(MyModel, validate_args=True, mymodel_does_not_contains_this_field=999) # this will␣
↪raise an error
```

Signals PRE_SAVE and POST_SAVE:

In very special cases a signal may facilitate implementing tests with DDF, but Django signals may not be satisfatory for testing pourposes because the developer does not have control of the execution order of the receivers. For this reason, DDF provides its own signals. It is possible to have only one receiver for each model, to avoid anti-patterns:

```
from django_dynamic_fixture import PRE_SAVE, POST_SAVE
def callback_function(instance):
    pass # do something
PRE_SAVE(MyModel, callback_function)
POST_SAVE(MyModel, callback_function)
```

Debugging

Print model instance values: P

Print all field values of an instance. You can also pass a list of instances or a *QuerySet*. It is useful for debug:

```
from django_dynamic_fixture import G, P

P(G(Model))
P(G(Model, n=2))
P(Model.objects.all())

# This will print some thing like that:
# :: Model my_app.MyModel (12345)
# id: 12345
# field_x: 'abc'
# field_y: 1
# field_z: 1
```

Debug Mode (New in 1.6.2)

Print debug log of DDF.

In settings.py:

```
DDF_DEBUG_MODE = True # default = False
```

In the test file:

```
G(MyModel, debug_mode=True)
```

List of Exceptions

- *UnsupportedFieldError*: DynamicFixture does not support this field.
- *InvalidConfigurationError*: The specified configuration for the field can not be applied or it is bugged.
- *InvalidManyToManyConfigurationError*: M2M attribute configuration must be a number or a list of DynamicFixture or model instances.
- *BadDataError*: The data passed to a field has some problem (not unique or invalid) or a required attribute is in ignore list.
- *InvalidCopierExpressionError*: The specified expression used in a Copier is invalid.
- *InvalidModelError*: Invalid Model: The class is not a model or it is abstract.
- *InvalidDDFSetupError*: ddf_setup.py has execution errors

CHAPTER 6

Patterns

Patterns

- Use *N* function for unit tests (not integration tests) for the main model.
- Use *G* function instead of *N* for shelving configurations. It helps to identify errors.
- Use *ignore_fields* option to deal with fields filled by listeners.
- Use custom values for unsupported fields.
- Use default shelve for unsupported fields with a custom function that generated data.
- Use *fill_nullable_fields* for unsupported nullable fields.
- Use *number_of_laps* option to test trees.
- [Automated Test Patterns](http://www.teses.usp.br/teses/disponiveis/45/45134/tde-02042012-120707/pt-br.php) (in Portuguese)

Anti-Patterns

- Using auto generated data in an assertion method.
- Shelving a static value for a field with *unique=True*. Raise an error.
- Overriding a shelved *Copier* with None.
- Using *Copier* to fix a bad design of the code.
- Too many named shelved configurations.
- Mix data fixture algorithms in the same test suite.
- Use shelve or named shelve configurations to avoid fixing a messy architecture.
- Extensive use of global set up: setup suite (from DDF), setup module and setup class (from unittest2).

A good automated test

- Simple
- Legible
- Repeatable
- Independent
- Isolated
- Useful
- Unique
- Accurate
- Professional
- Fast

CHAPTER 7

Nose plugins

DDF Setup Nose Plugin (New in 1.6.0)

This plugin create a "setup suite". It will load the file *ddf_setup.py* in the root of your Django project before to execute the first test. The file *ddf_setup.py* may contain shelved configurations or pythonic initial data. Django just load initial data from YAML, JSON, XML etc:

```
python manage.py test --with-ddf-setup
```

Queries Nose Plugin (New in 1.4.0)

This plugin print how many queries are executed by a test. It may be useful to determine performance issues of system features:

```
python manage.py test --with-queries
```

The following command print how many queries are executed when a model is saved. It may be useful to determine performance issues in overriding *save* methods and in extensive use of listeners *pre_save* and *post_save*:

```
python manage.py count_queries_on_save
```

CHAPTER 8

FileSystemDjangoTestCase (New in 1.3.0)

FileSystemDjangoTestCase has a tear down method to delete all test files generated by tests, to avoid letting trash in the file system:

```
class MyTests(FileSystemDjangoTestCase):
    def test_x(self):
        print(dir(self))
```

It also has a collection of methods to simplify tests that use files, like:

- create_temp_directory
- remove_temp_directory
- create_temp_file
- create_temp_file_with_name
- rename_temp_file
- remove_temp_file
- copy_file_to_dir
- add_text_to_file
- get_directory_of_the_file
- get_filename
- get_filepath
- get_content_of_file
- create_django_file_with_temp_file
- create_django_file_using_file

It also contains a set of assertion methods:

- assertFileExists

- assertFileDoesNotExists
- assertDirectoryExists
- assertDirectoryDoesNotExists
- assertDirectoryContainsFile
- assertDirectoryDoesNotContainsFile
- assertFilesHaveEqualLastModificationTimestamps
- assertFilesHaveNotEqualLastModificationTimestamps
- assertNumberOfFiles

CHAPTER 9

About

Change Log

Date format: yyyy/mm/dd

Version 1.9.5 - 2017/05/09

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.9.5>
- Bugfix: avoid GDALException on Django 1.11

Version 1.9.4 - 2017/04/17

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.9.4>
- Added support for django.contrib.postgres.fields.ArrayField field
- Fixed GeoDjango Point instantiation.

Version 1.9.3 - 2017/03/08

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.9.3>
- Improve compatibility for the DDF internal tests

Version 1.9.2 - 2017/03/08

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.9.2>
- Django 2.0 compatibility
- New: Support for wildcards *?* and * in the ignore fields

- Bugfix: Fixed DDF_TEST_GEODJANGO test issues

Version 1.9.1 - 2016/12/21

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.9.1>
- Bugfix: Django version parser
- Bugfix: NameError on invalid variable name

Version 1.9.0 - 2016/05/23

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.9.0>
- [New] Django 1.9 support
- [Bugfix] Fixed issue on ForeignKey field with default id
- [Bugfix] Fixed issue with SimpleUploadedFile

Version 1.8.4 - 2015/05/26

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.8.4>
- [New] UUIDField support
- [New] GeoDjango fields support (GeometryField, PointField, LineStringField, PolygonField, MultiPointField, MultiLineStringField, MultiPolygonField, GeometryCollectionField)
- [Update] Better error messages
- [Bugfix] BinaryField fixture fix
- [Update] Optimizations

Version 1.8.3 - 2015/05

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.8.3>
- [Update] No more deprecated methods

Version 1.8.2 - 2015/05

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.8.2>
- [New] Support for Django 1.8

Version 1.8.1 - 2014/12

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.8.1>
- [Update] Avoid conflicts with “instance” and “field” model field names.

Version 1.8.0 - 2014/09

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.8.0>
- [New] DDF_FIELD_FIXTURES global settings. You can include support of other fields here.
- [New] Support for BinaryField
- [Update] Django 1.7, Python 3.4 and Pypy official suppport (fixed some tests)
- [New] ReadTheDocs full documentation
- [Update] Fixed some print calls for python 3
- [Update] Nose plugin disable as default. Recommended behavior of nose plugins.
- [Update] ignore_fields parameter does not consider fields explicitly defined by the developer.
- [Update] Travis env using Tox

Version 1.7.0 - 2014/03/26

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.7.0>

Version 1.6.4 - 2012/12/30

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.6.4>
- [Bugfix] auto_now and auto_now_add must not be disabled forever (thanks for reporting)
- [New] Added global_sequential data fixture (Pull request, thanks)

Version 1.6.3 - 2012/04/10

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.6.3>
- [New] Pre save and post save special signals

Version 1.6.2 - 2012/04/09

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.6.2>
- [New] Debug Mode and option (global/local) to enable/disable debug mode

Version 1.6.1 - 2012/04/07

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.6.1>
- [New] New alias for F: field1__field2=value instead of field1=F(field2=value)
- [New] Named shelves inherit from default shelve

Version 1.6.0 - 2012/03/31

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.6.0>
- [New] Copier: option to copy a generated value for a field to another one. Useful for denormalizated fields.
- [New] Shelve/Library: option to store a default configuration of a specific model. Useful to avoid replicated code of fixtures. Global option: DDF_USE_LIBRARY.
- [New] Named Shelve: option to store multiple configurations for a model in the library.
- [New] Nose plugin for global set up.
- [New] P function now accept a queryset.

Version 1.5.1 - 2012/03/26

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.5.0>
- [New] global option: DDF_VALIDATE_ARGS that enable or disable field names.
- [Bugfix] F feature stop working.

Version 1.5.0 - 2012/03/25

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.5.0>
- [New] global settings: DDF_DEFAULT_DATA_FIXTURE, DDF_FILL_NULLABLE_FIELDS, DDF_IGNORE_FIELDS, DDF_NUMBER_OF_LAPS, DDF_VALIDATE_MODELS
- [New] new data fixture that generates random data
- [New] new data fixture that use sequential numbers only for fields that have unique=True
- [New] P function now accept a list of model instances
- [New] Option to call model_instance.full_clean() validation method before saving the object (DDF_VALIDATE_MODELS).
- [New] Validate field names. If a invalid field name is passed as argument, it will raise an InvalidConfigurationError exception.
- [Bugfix] DateField options 'auto_add_now' and 'auto_add' are disabled if a custom value is used.

Version 1.4.3 - 2012/02/23

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.4.3>
- [Bugfix] Bugfix in ForeignKeys with default values

Version 1.4.2 - 2011/11/07

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.4.2>
- [Bugfix] Bugfix in FileSystemDjangoTestCase

Version 1.4.1 - 2011/11/07

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.4.1>
- [New] Now you can set a custom File to a FileField and the file will be saved in the file storage system.
- **FileSystemDjangoTestCase**:
- [New] create_django_file_using_file create a django.File using the content of your file
- [New] create_django_file_with_temp_file now accepts a content attribute that will be saved in the generated file
- [Bugfix] now create_django_file_with_temp_file close the generated file

Version 1.4.0 - 2011/10/29

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.4.0>
- [New] Nose plugin to count queries on each test
- [New] Command line to count queries on the save (insert and update) of each model
- [Update] Field with choice and default must use the default value, not the first choice value
- [Update] Validation if the class is a models.Model instance
- [Update] Showing all stack trace, when an exception occurs
- **Decorators**:
- [Bugfix] default values of database engines were not used correctly
- **FileSystemDjangoTestCase**:
- [Testfix] Fixing tests

Version 1.3.1 - 2011/10/03

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.3.1>
- [Bugfix] Bugfixes in FileSystemDjangoTestCase

Version 1.3.0 - 2011/10/03

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.3.0>
- [New] File System Django Test Case
- [New] Decorators skip_for_database and only_for_database
- [Bugfix] Inheritance problems, before this version the DDF filled fields with the attribute parent_link

Version 1.2.3 - 2011/06/27

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.2.3>
- [Bugfix] string truncation to max_length

Version 1.2.2 - 2011/05/05

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.2.2>
- [Update] Improvements in exception messages

Version 1.2.1 - 2011/03/11

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.2.1>
- [Bugfix] Propagate ignored fields to self references
- [Refact] Refactoring

Version 1.2 - 2011/03/04

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.2>
- [New] ignore_fields
- [New] now it is possible to set the ID

Version 1.1

- <http://pypi.python.org/pypi/django-dynamic-fixture/1.0> (1.0 has the 1.1 package)
- [Bugfix] Bug fixes for 1.0

Version 1.0

- Initial version
- Ready to use in big projects

Collaborators

Paulo Cheque <http://twitter.com/paulocheque> <https://github.com/paulocheque>

Valder Gallo <http://valdergallo.com.br> <https://github.com/valdergallo>

Julio Netto <http://www.inerciasensorial.com.br> <https://bitbucket.org/inerte>

Pull Requests tips

About commit messages

- Messages in english only
- All messages have to follow the pattern: "[TAG] message"
- TAG have to be one of the following: new, update, bugfix, delete, refactoring, config, log, doc, mergefix

About the code

- One change (new feature, update, refactoring, bugfix etc) by commit
- All bugfix must have a test simulating the bug
- All commit must have 100% of test coverage

Running tests

Command:

```
python manage.py test --with-coverage --cover-inclusive --cover-html --cover-
↪package=django_dynamic_fixture.* --with-queries --with-ddf-setup
```

TODO list

Tests and Bugfixes

- with_queries bugfixes (always print 0 queries)
- Deal with relatioships with dynamic related_name
- bugfix in fdf or ddf: some files/directories are not deleted
- tests with files in ddf
- tests with proxy models
- tests with GenericRelations, GenericForeignKey etc
- more tests with OneToOneField(parent_link=True)
- Test with python 2.4
- Test with python 2.5
- Test with python 3.*

Features

- auto config of denormalizated fields
- related_name documentation or workaround
- today, yesterday, tomorrow on fdf
- string generation according to a regular expression

Documentation

- with_queries documentation
- example to generate models with validators in fields or in clean methods

CHAPTER 10

Indices and tables

- genindex
- modindex
- search