fillmydb Documentation

Release 0.1.0

Vlad Calin

September 14, 2016

Contents

Contents:

CHAPTER 1

Introduction

1.1 Description

The `fillmydb` library aims to help you with quickly populating your database with nice-looking data with as few lines of code as possible. Basically, all you have to do is to wrap your models in a *`fillmydb.ModelWrapper`*, assign to each field an instance of *`fillmydb.FieldSpec`* in order to determine how each filed should look like and then generate instances. The foreign keys are handled automatically.

The supported model types so far are:

- peewee
- django

Warning: At this moment, circular dependencies are not treated and may break the generation algorithm.

The generation algorithm simplified is the following:

```
initially orderes the queue
marks all models as unresovled
while queue not empty:
    picks the next model with no unresolved dependencies
    for each instance:
        resolves each field in the model:
            if foreign key, picks a random instance of the referenced model
            otherwise, resolves as usual
        saves to database
    marks model as processed
```

In the next section will be described how to use the module

1.2 References

- fake-factory - generates good-looking test data
- peewee - a nice ORM for SQLite, PostgreSQL, MySQL and others.
- django - a full-stack web framework.

CHAPTER 2

Usage

Value generation consists in the following steps:

> **Warning:** If you plan on using wrapping Django models, you must call the *fillmydb.initialize_django()* function **before importing your models**.

1. Wrap your models in a *fillmydb.ModelWrapper* like this:

```
wrapper = ModelWrapper(MyFirstModel, MySecondModel, ...)
```

2. Specify how the fields should look like using *fillmydb.FieldSpec* instances:

```
wrapper[MyFirstModel].first_field = FieldSpec(generate_func1)
wrapper[MyFirstModel].second_field = FieldSpec(generate_func2)
...

wrapper[MySecondModel].first_field = FieldSpec(generate_for_model2)
wrapper[MySecondModel].second_field = FieldSpec(generate_for_model2_again)
...
```

3. Generate how many instances you need directly into your database:

```
wrapper.generate(1000, 2000, ...)
```

A complete example

Lets assume we have the following peewee models:

```
class User(peewee.Model):
    name = peewee.CharField()
    username = peewee.CharField()
    password_hash = peewee.BlobField()
    email = peewee.CharField()
    visits = peewee.IntegerField()
    description = peewee.CharField()


class Post(peewee.Model):
    title = peewee.CharField()
    text = peewee.CharField()
    by_user = peewee.ForeignKeyField(User)


class Like(peewee.Model):
```

```
    by_user = peewee.ForeignKeyField(User)
    to_post = peewee.ForeignKeyField(Post)
```

Now let's generate our instances:

```
wrapper = ModelWrapper(User, Post, Like)  # order doesn't matter
factory = faker.Factory.create()  # we use the fake-factory module generate nice-looking data

wrapper[User].name = FieldSpec(factory.name)
wrapper[User].username = FieldSpec(factory.user_name)
wrapper[User].description = FieldSpec(factory.text)
wrapper[User].password_hash = FieldSpec(factory.binary, length=25)
wrapper[User].email = FieldSpec(factory.email)
wrapper[User].visits = FieldSpec(factory.pyint)

wrapper[Post].title = FieldSpec(faker.sentence)
wrapper[Post].text = FieldSpec(factory.text)

wrapper.generate(100, 200, 300)
```

Now we have 100 fresh instances of `User`, 200 instances of `Post` and 300 instances of `Like`

CHAPTER 3

The fillmydb module

class fillmydb.**ModelWrapper**(**models*)

Creates a wrapper around the *models* models that must be of the same type.

Parameters **models** – The models to be wrapped and used afterwards for generating instances.

Raises **ValueError** – when the models are not of the same type (belong to different ORMs)

generate(**counts*)

Generates and persists items. *counts* is a list of integers that indicate how many instances of each model should generate. The order is preserved from the models specified in constructor.

For example, if `wrapper` was declared as `ModelWrapper(Model1, Model2, Model3)` the following code

```
wrapper.generate(10, 20, 15)
```

will generate 10 instances of `Model1`, 20 instances of `Model2` and 15 instances of `Model3`.

Parameters **counts** – the quantity of each item to be generated.

Returns None

class fillmydb.**FieldSpec**(*func*, **args*, ***kwargs*)

The generation logic for mock values for fields.

The mocked values of the fields will be generated as `func(*args, **kwargs)`.

Parameters

- **func** – a callable object.
- **args** – the ordinal parameters `func` will be called with.
- **kwargs** – the named parameters `func` will be called with.

resolve()

Method that returns the generated value.

Returns an object that will be directly assigned to the field when instantiating a model.

fillmydb.**initialize_django**(*settings_py_path*)

Initializes the environment required by Django. Must be called **before** importing your models:

```
import fillmydb
fillmydb.initialize_django("path/to/settings.py")

from mydjangoproject.myapp.models import MyModel, MyOtherModel
```

```
    ...
```

Parameters **settings_py_path** – Path to the `settings.py` file from your Django project.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

F

G

I

M

R