pytest-factoryboy Documentation

Release 2.1.0

Oleg Pidsadnyi, Anatoly Bubenkov and others

Apr 27, 2021

Contents

Contents

CHAPTER 1

factory_boy integration with the pytest runner

pytest-factoryboy makes it easy to combine `factory` approach to the test setup with the `dependency` injection, heart of the pytest fixtures.

1.1 Install pytest-factoryboy

```
pip install pytest-factoryboy
```

1.2 Concept

Library exports a function to register factories as fixtures. Fixtures are contributed to the same module where register function is called.

1.3 Factory Fixture

Factory fixtures allow using factories without importing them. Name convention is lowercase-underscore class name.

```
import factory
from pytest_factoryboy import register

class AuthorFactory(factory.Factory):
```

(continues on next page)

(continued from previous page)

```
    class Meta:
        model = Author


register(AuthorFactory)


def test_factory_fixture(author_factory):
    author = author_factory(name="Charles Dickens")
    assert author.name == "Charles Dickens"
```

1.4 Model Fixture

Model fixture implements an instance of a model created by the factory. Name convention is model's lowercase-underscore class name.

```
import factory
from pytest_factoryboy import register

@register
class AuthorFactory(factory.Factory):
    class Meta:
        model = Author

    name = "Charles Dickens"


def test_model_fixture(author):
    assert author.name == "Charles Dickens"
```

Model fixtures can be registered with specific names. For example if you address instances of some collection by the name like "first", "second" or of another parent as "other":

```
register(BookFactory)  # book
register(BookFactory, "second_book")  # second_book

register(AuthorFactory) # author
register(AuthorFactory, "second_author") # second_author

register(BookFactory, "other_book")  # other_book, book of another author

@pytest.fixture
def other_book__author(second_author):
    """Make the relation of the second_book to another (second) author."""
    return second_author
```

1.5 Attributes are Fixtures

There are fixtures created for factory attributes. Attribute names are prefixed with the model fixture name and double underscore (similar to the convention used by factory_boy).

```
@pytest.mark.parametrize("author__name", ["Bill Gates"])
def test_model_fixture(author):
    assert author.name == "Bill Gates"
```

1.6 SubFactory

Sub-factory attribute points to the model fixture of the sub-factory. Attributes of sub-factories are injected as dependencies to the model fixture and can be overridden via the parametrization.

1.7 Related Factory

Related factory attribute points to the model fixture of the related factory. Attributes of related factories are injected as dependencies to the model fixture and can be overridden via the parametrization.

1.8 post-generation

Post-generation attribute fixture implements only the extracted value for the post generation function.

1.9 Integration

An example of factory_boy and pytest integration.

factories/__init__.py:

```
import factory
from faker import Factory as FakerFactory

faker = FakerFactory.create()


class AuthorFactory(factory.django.DjangoModelFactory):
    """Author factory."""

    name = factory.LazyAttribute(lambda x: faker.name())

    class Meta:
        model = 'app.Author'


class BookFactory(factory.django.DjangoModelFactory):
    """Book factory."""

    title = factory.LazyAttribute(lambda x: faker.sentence(nb_words=4))

    class Meta:
        model = 'app.Book'

    author = factory.SubFactory(AuthorFactory)
```

tests/conftest.py:

```
from pytest_factoryboy import register

from factories import AuthorFactory, BookFactory

register(AuthorFactory)
register(BookFactory)
```

tests/test_models.py:

```
from app.models import Book
from factories import BookFactory

def test_book_factory(book_factory):
    """Factories become fixtures automatically."""
    assert isinstance(book_factory, BookFactory)

def test_book(book):
    """Instances become fixtures automatically."""
    assert isinstance(book, Book)

@pytest.mark.parametrize("book__title", ["PyTest for Dummies"])
@pytest.mark.parametrize("author__name", ["Bill Gates"])
def test_parametrized(book):
    """You can set any factory attribute as a fixture using naming convention."""
    assert book.name == "PyTest for Dummies"
    assert book.author.name == "Bill Gates"
```

1.10 Fixture partial specialization

There is a possibility to pass keyword parameters in order to override factory attribute values during fixture registration. This comes in handy when your test case is requesting a lot of fixture flavors. Too much for the regular pytest parametrization. In this case you can register fixture flavors in the local test module and specify value deviations inside `register` function calls.

```
register(AuthorFactory, "male_author", gender="M", name="John Doe")
register(AuthorFactory, "female_author", gender="F")


@pytest.fixture
def female_author__name():
    """Override female author name as a separate fixture."""
    return "Jane Doe"


@pytest.mark.parametrize("male_author__age", [42])  # Override even more
def test_partial(male_author, female_author):
    """Test fixture partial specialization."""
    assert male_author.gender == "M"
    assert male_author.name == "John Doe"
    assert male_author.age == 42

    assert female_author.gender == "F"
    assert female_author.name == "Jane Doe"
```

1.11 Fixture attributes

Sometimes it is necessary to pass an instance of another fixture as an attribute value to the factory. It is possible to override the generated attribute fixture where desired values can be requested as fixture dependencies. There is also a lazy wrapper for the fixture that can be used in the parametrization without defining fixtures in a module.

LazyFixture constructor accepts either existing fixture name or callable with dependencies:

```
import pytest
from pytest_factoryboy import register, LazyFixture


@pytest.mark.parametrize("book__author", [LazyFixture("another_author")])
def test_lazy_fixture_name(book, another_author):
    """Test that book author is replaced with another author by fixture name."""
    assert book.author == another_author


@pytest.mark.parametrize("book__author", [LazyFixture(lambda another_author: another_
↪author)])
def test_lazy_fixture_callable(book, another_author):
    """Test that book author is replaced with another author by callable."""
    assert book.author == another_author


# Can also be used in the partial specialization during the registration.
register(BookFactory, "another_book", author=LazyFixture("another_author"))
```

CHAPTER 2

Post-generation dependencies

Unlike factory_boy which binds related objects using an internal container to store results of lazy evaluations, pytest-factoryboy relies on the PyTest request.

Circular dependencies between objects can be resolved using post-generation hooks/related factories in combination with passing the SelfAttribute, but in the case of PyTest request fixture functions have to return values in order to be cached in the request and to become available to other fixtures.

That's why evaluation of the post-generation declaration in pytest-factoryboy is deferred until calling the test funciton. This solves circular dependecy resolution for situations like:

```
o->[ A ]-->[ B ]<--[ C ]-o
|                        |
o----(C depends on A)----o
```

On the other hand deferring the evaluation of post-generation declarations evaluation makes their result unavailable during the generation of objects that are not in the circular dependecy, but they rely on the post-generation action.

pytest-factoryboy is trying to detect cycles and resolve post-generation dependencies automatically.

```
from pytest_factoryboy import register


class Foo(object):
    def __init__(self, value):
        self.value = value


class Bar(object):
    def __init__(self, foo):
        self.foo = foo


@register
class FooFactory(factory.Factory):
    """Foo factory."""
```

(continues on next page)

(continued from previous page)

```
    class Meta:
        model = Foo

    value = 0

    @factory.post_generation
    def set1(foo, create, value, **kwargs):
        foo.value = 1


class BarFactory(factory.Factory):
    """Bar factory."""

    foo = factory.SubFactory(FooFactory)

    @classmethod
    def _create(cls, model_class, foo):
        assert foo.value == 1  # Assert that set1 is evaluated before object␣
↪generation
        return super(BarFactory, cls)._create(model_class, foo=foo)

    class Meta:
        model = Bar


register(
    BarFactory,
    'bar',
)
"""Forces 'set1' to be evaluated first."""


def test_depends_on_set1(bar):
    """Test that post-generation hooks are done and the value is 2."""
    assert depends_on_1.foo.value == 1
```

2.1 Hooks

pytest-factoryboy exposes several pytest hooks which might be helpful for e.g. controlling database transaction, for reporting etc:

- pytest_factoryboy_done(request) - Called after all factory based fixtures and their post-generation actions have been evaluated.

2.2 License

This software is licensed under the MIT license.

CHAPTER 3

Authors

Oleg Pidsadnyi original idea and implementation

These people have contributed to *pytest-factoryboy*, in alphabetical order:

- Anatoly Bubenkov
- Daniel Duong
- Daniel Hahler
- Hugo van Kemenade
- p13773
- Vasily Kuznetsov

CHAPTER 4

Changelog

4.1 Unreleased

4.2 2.1.0

- Add support for factory_boy >= 3.2.0
- Drop support for Python 2.7, 3.4, 3.5. We now support only python >= 3.6.
- Drop support for pytest < 4.6. We now support only pytest >= 4.6.
- Add missing versions of python (3.9 and 3.10) and pytest (6.x.x) to the CI test matrix.

4.3 2.0.3

- Fix compatibility with pytest 5.

4.4 2.0.2

- Fix warning *use of getfuncargvalue is deprecated, use getfixturevalue* (sliverc)

4.5 2.0.1

Breaking change due to the heavy refactor of both pytest and factory_boy.

- Failing test for using a *attributes* field on the factory (blueyed)
- Minimal pytest version is 3.3.2 (olegpidsadnyi)

- Minimal factory_boy version is 2.10.0 (olegpidsadnyi)

4.6 1.3.2

- use {posargs} in pytest command (blueyed)
- pin factory_boy<2.9 (blueyed)

4.7 1.3.1

- fix LazyFixture evaluation order (olegpidsadnyi)

4.8 1.3.0

- replace request._fixturedefs by request._fixture_defs (p13773)

4.9 1.2.2

- fix post-generation dependencies (olegpidsadnyi)

4.10 1.2.1

- automatical resolution of the post-generation dependencies (olegpidsadnyi, kvas-it)

4.11 1.1.6

- fixes fixture function module name attribute (olegpidsadnyi)
- fixes _after_postgeneration hook invocation for deferred post-generation declarations (olegpidsadnyi)

4.12 1.1.5

- support factory models to be passed as strings (bubenkoff)

4.13 1.1.3

- circular dependency determination is fixed for the post-generation (olegpidsadnyi)

4.14 1.1.2

- circular dependency determination is fixed for the RelatedFactory attributes (olegpidsadnyi)

4.15 1.1.1

- fix installation issue when django environment is not set (bubenkoff, amakhnach)

4.16 1.1.0

- fixture dependencies on deferred post-generation declarations (olegpidsadnyi)

4.17 1.0.3

- post_generation extra parameters fixed (olegpidsadnyi)
- fixture partial specialization (olegpidsadnyi)
- fixes readme and example (dduong42)
- lazy fixtures (olegpidsadnyi)
- deferred post-generation evaluation (olegpidsadnyi)
- hooks (olegpidsadnyi)

4.18 1.0.2

- refactoring of the fixture function compilation (olegpidsadnyi)
- related factory fix (olegpidsadnyi)
- post_generation fixture dependency fixed (olegpidsadnyi)
- model fixture registration with specific name (olegpidsadnyi)
- README updated (olegpidsadnyi)

4.19 1.0.1

- use `inflection` package to convert camel case to underscore (bubenkoff)

4.20 1.0.0

- initial release (olegpidsadnyi)