SQLAlchemy Model Factory

Release 0.1.0

Dan Cardin

Apr 19, 2023

CONTENTS:

1 Quickstart **1**
- 1.1 Installation . . . 1
- 1.2 Usage . . . 1

2 Testing **3**
- 2.1 Pytest . . . 3

3 Factories **5**
- 3.1 Basic . . . 5
- 3.2 General Use Functions . . . 6
- 3.3 Sources of Uniqueness . . . 6
- 3.4 Fluency . . . 7

4 Declarative API **9**
- 4.1 Conversion from `@register_at` . . . 10

5 Options **13**
- 5.1 Factory-level Options . . . 13
- 5.2 Call-level Options . . . 13

6 API **15**
- 6.1 Declarative . . . 15
- 6.2 Factory Utilities . . . 16
- 6.3 Pytest Plugin . . . 18

7 Indices and tables **19**

Python Module Index **21**

Index **23**

CHAPTER

ONE

QUICKSTART

sqlalchemy-model-factory aims to make it easy to write factory functions for sqlalchemy models, particularly for use in testing.

It should make it easy to define as many factories as you might want, with as little boilerplate as possible, while remaining as unopinionated as possible about the behavior going in your factories.

1.1 Installation

```
pip install sqlalchemy-model-factory
```

1.2 Usage

Suppose you've defined a `Widget` model, and for example you want to test some API code that queries for `Widget` instances. Couple of factory functions might look like so:

```
# tests/test_example_which_uses_pytest
from sqlalchemy_model_factory import autoincrement, register_at
from . import models

@register_at('widget')
def new_widget(name, weight, color, size, **etc):
    """My goal is to allow you to specify *all* the options a widget might require.
    """
    return Widget(name, weight, color, size, **etc)

@register_at('widget', name='default')
@autoincrement
def new_default_widget(autoincrement=1):
    """My goal is to give you a widget with as little input as possible.
    """
    # I'm gonna call the other factory function...because i can!
    return new_widget(
```

(continues on next page)

(continued from previous page)

```
        f'default_name{autoincrement}',
        weight=autoincrement,
        color='rgb({0}, {0}, {0})'.format(autoincrement),
        size=autoincrement,
    )
```

What this does, is register those functions to the registry of factory functions, within the "widget" namespace, at the `name` (defaults to `new`) location in the namespace.

So when I go to write a test, all I need to do is accept the `mf` fixture (and lets say a `session` db connection fixture to make assertions against) and I can call all the factories that have been registered.

```
def test_example_model(mf, session):
    widget1 = mf.widget.new('name', 1, 'rgb(0, 0, 0)', 1)
    widget2 = mf.widget.default()
    widget3 = mf.widget.default()
    widget4 = mf.widget.default()

    widgets = session.query(Widget).all()
    assert len(widgets) == 4
    assert widgets[0].name == 'name'
    assert widgets[1].id == widget2.id
    assert widgets[2].name == widget3.name
    assert widgets[3].color == 'rgb(3, 3, 3)'
```

In a simple toy example, where you don't gain much on the calls themselves the benefits are primarily:

- The instances are automatically put into the database and cleaned up after the test.
- You can make assertions without hardcoding the values, because you get back a handle on the object.

But as the graph of models required to set up a particular scenario grows:

- You can define factories as complex as you want
  - They can create related objects and assign them to relationships
  - They can be given sources of randomness or uniqueness to not violate constraints
  - They can compose with eachother (when called normally, they're the same as the original function).

CHAPTER

TWO

TESTING

A (or maybe **the**) primary usecase for this package is is in writing test tests concisely, ergonmically, and readably.

To that end, we integrate with the testing framework in order to provide good UX in your tests.

2.1 Pytest

We provide default implementations of a couple of pytest fixtures: `mf_engine`, `mf_session`, and `mf_config`. However this assumes you're okay running your code as though it's executed in SQLite, and with default session parameters.

If your system will work under those conditions, great! Simply go on and use the *mf* fixture which gives you a handle on a `ModelFactory`

```
from sqlalchemy_model_factory import registry

@registry.register_at('foo')
def new_foo():
    return Foo()

def test_foo(mf):
    foo = mf.foo.new()
    assert isinstance(foo, Foo)
```

If, however, you make use of feature not available in SQLite, you may need a handle on a real database engine. Supposing you've got a postgres database available at `db:5432`, you can put the following into your `tests/conftest.py`.

```
import pytest
from sqlalchemy import create_engine

@pytest.fixture
def mf_engine():
    return create_engine('psycopg2+postgresql://db:5432')

# now the `mf` fixture should work
```

Furthermore, if your application works in a context where you assume your `session` has particular options set, you can similarly plug in your own session.

```
import pytest
from sqlalchemy.orm.session import sessionmaker
```

(continues on next page)

(continued from previous page)

```
@pytest.fixture
def mf_session(mf_engine):
    Session = sessionmaker()  # Set your options
    return Session(bind=engine)

# now the `mf` fixture should work
```

Finally, there are a set of hooks through which you can configure the behavior of the `ModelFactory` itself through the `mf_config` fixture. If defined, this fixture should return a `dict`, the contents of which would be the config options available.

Below is defined, the equivalent of a maximally defined `mf_config` fixture with all the values set to their defaults. **Note** That as a user, you only need to include options which you want overridden from their defaults.

```
@pytest.fixture
def mf_config():
    return {
        # Whether the calling of all factory functions should commit, or just flush.
        "commit": True,

        # Whether the actions performed by the model-factory should attempt to revert.␣
↪Certain
        # test circumstances (like complex relationships, or direct sql `execute` calls␣
↪might
        # mean cleanup will fail an otherwise valid test.
        "cleanup": True,
    }
```

CHAPTER

THREE

FACTORIES

3.1 Basic

So the most basic factories that you can write are just functions which return models.

```
from sqlalchemy_model_factory import register_at

@register_at('foo')
def new_foo():
    return Foo()

def test_foo(mf):
    foo = mf.foo.new()
    assert isinstance(foo, Foo)
```

3.1.1 Nested

Note, you can also create nested models through relationships and whatnot; and that will all work normally.

```
from sqlalchemy_model_factory import register_at

class Foo(Base):
    ...

    bar = relationship('Bar')

class Bar(Base):
    ...

    baz = relationship('Baz')

@register_at('foo')
def new_foo():
    ...
```

3.2 General Use Functions

In some cases, you'll have a a function already handy that returns the equivalent signature of a model (or you just **want** a function that returns the signature).

In this case, your function will act as the originally defined function when called normally, however when invoked in the context of the `ModelFactory`, it returns the specified model instance.

```
from sqlalchemy_model_factory import for_model, register_at

@register_at('foo')
@for_model(Foo)
def new_foo():
    return {'id': 1, 'name': 'bar'}

def test_foo(mf):
    foo = mf.foo.new()
    assert foo.id == 1
    assert foo.name == 'bar'

    raw_foo = new_foo()
    assert raw_foo == {'id': 1, 'name': 'bar'}
```

3.3 Sources of Uniqueness

Suppose you've got a column defined with a constraint, like *name = Column(types.Integer(), unique=True)*.

Suddenly you'll need to parametrize your factory to accept a `name` param. However if you **actually** don't care about the specific name values, you have a few options.

3.3.1 Autoincrement

Automatically incrementing number values is one option. Your factory will be automatically supplied with an `autoincrement` parameter, known to not collide with previously generated values.

```
from sqlalchemy_model_factory import autoincrement, register_at

@register_at('foo')
@autoincrement
def new_foo(autoincrement=1):
    return Foo(name=f'name{autoincrement}')

def test_foo(mf):
    assert mf.foo.new().name == 'name1'
    assert mf.foo.new().name == 'name2'
    assert mf.foo.new().name == 'name3'
```

3.4 Fluency

You've been working along and writing factories and you finally find yourself in a situation like this.

```
@register_at('foo')
@autoincrement
def new_foo(name='name', height=2, width=3, depth=3, category='foo', autoincrement=1):
    ...
```

And in the event your test requires a number of identical parameters across multiple calls, you might end up with test code that looks like.

```
def test_foo(mf):
    width_4 = mf.foo.new(height=3, category='bar', width=4)
    width_5 = mf.foo.new(height=3, category='bar', width=5)
    width_6 = mf.foo.new(height=3, category='bar', width=6)
    width_7 = mf.foo.new(height=3, category='bar', width=7)
    ...
```

The above (as a dirt simple example, that might be easily solved in different ways) has got **most** of its information duplicated unnecessarily.

3.4.1 The "fluent" decorator

A simple solution to this general problem category is the `fluent` decorator. Which adapts a given callable to be able to be called in a fluent style.

```
def test_foo(mf):
    bar_type_foo = mf.foo.new(3).category('bar')

    width_4 = bar_type_foo.width(4).bind()
    width_5 = bar_type_foo.width(5).bind()
    width_6 = bar_type_foo.width(6).bind()
    width_7 = bar_type_foo.width(7).bind()
```

Now in this particular case, you could have just done a for-loop over the original set of calls, or maybe `functools.partial` could have sufficed, but the fluent pattern is more generally useful than just in cases like this.

Also from the callee's perspective, there's not necessarily any requirement that all the args have their parameter names supplied, so you might end up reading `foo.new('a', 3, 4, 5, 'ro')`, which is arguably far less readable.

To note, the `bind` call at the end of each expression above is necessary to let the fluent calls know that its done being called (because as you might notice, we didn't call all the available methods we could have called). But this also serves as a convenient point at which to add custom behaviors. (for example you *could* supply *.bind(call_after=print)* to have it print out the final result of the function; see the api portion of the docs for the full set of options.)

3.4.2 Class-style factories

From the perspective of the model factory, all factory "functions" are just callables, so you can always manually mimic something like the above `fluent` decorator in a class so that you can implement your own custom behavior for each option.

```
@register_at('foo')
class NewFoo:
    def __init__(self, **kwargs):
        self.kwargs

    def name(self, name):
        self.__class__(**self.kwargs, name=name)

    def width(self, width):
        self.__class__(**self.kwargs, width=width)

    def bind(self):
        return Foo(**self.kwargs)
```

Albeit, with the above, very naive implementation, your test code would end up looking like

```
def test_foo(mf):
    bar_foo = mf.foo.new().name('bar')
    width_4 = bar_fo.width(4).bind()
    width_5 = bar_fo.width(5).bind()
```

CHAPTER

FOUR

DECLARATIVE API

There are a few benefits to declaratively specifying the factory function tree:

- The vanilla `@register_at` decorator is dynamic and string based, which leaves no direct import or path to trace back to the implementing function from a `ModelFactory` instance.
- There is, perhaps, an alternative declarative implementation which ensures the yielded `ModelFactory` literally **is** the heirarchy specified.
  - As is, if you're in a context in which you can type annotate the model factory, then this enables typical LSP features like "Go to Definition".

    In the most common cases, such as with `pytest`, you'll be being handed an untyped `mf` (model_factory) fixture instance. Here, you can type annotate the argument as being your `@declarative` ly decorated class.

```
# some_module.py
class other:
    def new():
        ...

# factory.py
from some_module import other

@declarative
class ModelFactory:
    other = other

    class namespace:
        def example():
            ...

# tests.py
from sqlalchemy_model_factory.pytest import create_registry_fixture
from factory import ModelFactory

mf_registry = create_registry_fixture(ModelFactory)

# `mf` being a sqlalchemy_model_factory-provided fixture.
def test_factory(mf: ModelFactory):
   ...
```

Alternatively, a registry can be provided to the decorator directly, ou have one pre-constructed.

```
registry = Registry()

@declarative(registry=registry)
class ModelFactory:
    def fn():
        ...
```

Note: interior classes to the decorator, including both the root class, as well as any nested class or attributes which are raw types, will be instantiated. This is notable, primarily in the event that an *__init__* is defined on the class for whatever reason. Each class will be instantiated once without arguments.

4.1 Conversion from `@register_at`

If you have an existing body of model-factory functions registered using the `@register_at` pattern, you can incrementally adopt (and therefore incrementally get viable type hinting support) the declarative api.

If you are importing `from sqlalchemy_model_factory import register_at`, today, you can import `from sqlalchemy_model_factory import registry`, and send that into the `@declarative` decorator:

```
from sqlalchemy_model_factory import registry

@declarative(registry=registry)
class ModelFactory:
    def example():
        ...
```

Alternatively, you can switch to manually constructing your own `Registry`, though you will need to change your `@register_at` calls to use it!

```
from sqlalchemy_model_factory import Registry, declarative

registry = Registry()

@register_at("path", name="new")
def new_path():
    ...

@declarative(registry=registry)
class Base:
    def example():
        ...
```

Then once you make use of the annotation, say in some test:

```
def test_path(mf: Base):
    mf.example()
```

you should get go-to-definition and hinting support for the declaratively specified methods **only**.

Note: You might see mypy type errors like `Type[...] has no attribute "..."` for `@register_at`. You can

either ignore these, or else apply the `compat` as a superclass to your declarative:

```
from sqlalchemy_model_factory import declarative

@declarative.declarative
class Factory(declarative.compat):
    ...
```

CHAPTER

FIVE

OPTIONS

5.1 Factory-level Options

Options can be supplied to `register_at` at the factory level to alter the default behavior when calling a factory.

Factory-level options include:

- commit: `True/False` (default `True`)

  Whether the given factory should commit the models it produces.

- merge: `True/False` (default `False`)

  Whether the given factory should `Session.add` the models it produces, or `Session.merge` them.

  This option can be useful for obtaining a reference to some model you **know** is already in the database, but you dont currently have a handle on.

For example:

```
@register_at("widget", name="default", merge=True)
def default_widget():
    return Widget()
```

5.2 Call-level Options

All options available at the factory-level can also be provided at the call-site when calling the factories, although their arguments are postfixed with a trailing _ to avoid colliding with normal factory arguments.

```
def test_widget(mf):
    widget = mf.widget.default(merge_=True, commit_=True)
```

CHAPTER

SIX

API

6.1 Declarative

class sqlalchemy_model_factory.declarative.**DeclarativeMF**

Provide an alternative to the class decorator for declarative base factories.

Today there's no meaningful difference between the decorator and the subclass method, except the interface.

Examples

```
>>> class ModelFactory(DeclarativeMF):
...     def default(id: int = None):
...         return Foo(id=id)
```

or

```
>>> registry = Registry()
>>>
>>> class ModelFactory(DeclarativeMF, registry=registry):
...     def default(id: int = None):
...         return Foo(id=id)
```

sqlalchemy_model_factory.declarative.**declarative**(*_cls=None*, *, *registry=None*)

Decorate a base object on which factory functions reside.

The primary benefit of declaratively specifying the factory function tree is that it enables references to the factory to be type aware, enabling things like "Go to Definition".

Note interior classes to the decorator, including both the root class, as well as any nested class or attributes which are raw types, will be instantiated. This is notable, primarily in the event that an *__init__* is defined on the class for whatever reason. Each class will be instantiated once without arguments.

Examples

```
>>> @declarative
... class ModelFactory:
...     class namespace:
...         def fn():
...             ...
```

```
>>> from sqlalchemy_model_factory.pytest import create_registry_fixture
>>> mf_registry = create_registry_fixture(ModelFactory)
```

```
>>> def test_factory(mf: ModelFactory):
...     ...
```

Alternatively, a registry can be provided to the decorator directly, if you have one pre-constructed.

```
>>> registry = Registry()
>>>
>>> @declarative(registry=registry)
... class ModelFactory:
...     def fn():
...         ...
```

Note due to the dynamic nature of fixtures, you must annotate the fixture argument to have the type of the root, declarative class.

6.2 Factory Utilities

`sqlalchemy_model_factory.utils.`**`autoincrement`**(*fn=None*, *, *start=1*)

Decorate registered callables to provide them with a source of uniqueness.

Parameters

- **fn** (`Optional[Callable]`) – The callable
- **start** (`int`) – The starting number of the sequence to generate

Examples

```
>>> @autoincrement
... def new(autoincrement=1):
...     return autoincrement
>>> new()
1
>>> new()
2
```

```
>>> @autoincrement(start=4)
... def new(autoincrement=1):
...     return autoincrement
>>> new()
```

(continues on next page)

(continued from previous page)

```
4
>>> new()
5
```

class sqlalchemy_model_factory.utils.**fluent**(*fn*, *signature=None*, *pending_args=None*)

Decorate a function with *fluent* to enable it to be called in a "fluent" style.

Examples

```
>>> @fluent
... def foo(a, b=None, *args, c=3, **kwargs):
...     print(f'(a={a}, b={b}, c={c}, args={args}, kwargs={kwargs})')
```

```
>>> foo.kwargs(much=True, surprise='wow').a(4).bind()
(a=4, b=None, c=3, args=(), kwargs={'much': True, 'surprise': 'wow'})
```

```
>>> foo.args(True, 'wow').a(5).bind()
(a=5, b=None, c=3, args=(True, 'wow'), kwargs={})
```

```
>>> partial = foo.a(1)
>>> partial.b(5).bind()
(a=1, b=5, c=3, args=(), kwargs={})
```

```
>>> partial.b(6).bind()
(a=1, b=6, c=3, args=(), kwargs={})
```

bind(*, *call_before=None*, *call_after=None*)

Finalize the call chain for a fluently called factory.

Parameters

- **call_before** (Optional[Callable]) – When provided, calls the given callable, supplying the args and kwargs being sent into the factory function before actually calling it. If the *call_before* function returns anything, the 2-tuple of (args, kwargs) will be replaced with the ones passed into the *call_before* function.
- **call_after** (Optional[Callable]) – When provided, calls the given callable, supplying the result of the factory function call after having called it. If the *call_after* function returns anything, the result of *call_after* will be replaced with the result of the factory function.

sqlalchemy_model_factory.utils.**for_model**(*typ*)

Decorate a factory that returns a *Mapping* type in order to coerce it into the *typ*.

This decorator is only invoked in the context of model factory usage. The intent is that a factory function could be more generally useful, such as to create API inputs, that also happen to correspond to the creation of a model when invoked during a test.

Examples

```
>>> class Model:
...     def __init__(self, **kwargs):
...         self.kw = kwargs
...
...     def __repr__(self):
...         return f"Model(a={self.kw['a']}, b={self.kw['b']}, c={self.kw['c']})"
```

```
>>> @for_model(Model)
... def new_model(a, b, c):
...     return {'a': a, 'b': b, 'c': c}
```

```
>>> new_model(1, 2, 3)
{'a': 1, 'b': 2, 'c': 3}
>>> new_model.for_model(1, 2, 3)
Model(a=1, b=2, c=3)
```

6.3 Pytest Plugin

A pytest plugin as a simplified way to use the ModelFactory.

General usage requires the user to define either a *mf_engine* or a *mf_session* fixture. Once defined, they can have their tests depend on the exposed *mf* fixture, which should give them access to any factory functions on which they've called *register_at*.

`sqlalchemy_model_factory.pytest.`**`mf`**`(`*mf_registry*, *mf_session*, *mf_config*`)`

Define a fixture for use of the ModelFactory in tests.

`sqlalchemy_model_factory.pytest.`**`mf_config`**`()`

Define a default fixture in for the model factory configuration.

`sqlalchemy_model_factory.pytest.`**`mf_engine`**`()`

Define a default fixture in for the database engine.

`sqlalchemy_model_factory.pytest.`**`mf_registry`**`()`

Define a default fixture for the general case where the default registry is used.

`sqlalchemy_model_factory.pytest.`**`mf_session`**`(`*mf_engine*`)`

Define a default fixture in for the session, in case the user defines only *mf_engine*.

`class` `sqlalchemy_model_factory.pytest.`**`pytest`**

Guard against pytest not being installed.

The below function will simply act as a normal function if pytest is not installed.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

sqlalchemy_model_factory, 15
sqlalchemy_model_factory.declarative, 15
sqlalchemy_model_factory.pytest, 18
sqlalchemy_model_factory.utils, 16

INDEX

A

autoincrement() (*in module sqlalchemy_model_factory.utils*), 16

B

bind() (*sqlalchemy_model_factory.utils.fluent method*), 17

D

declarative() (*in module sqlalchemy_model_factory.declarative*), 15
DeclarativeMF (*class in sqlalchemy_model_factory.declarative*), 15

F

fluent (*class in sqlalchemy_model_factory.utils*), 17
for_model() (*in module sqlalchemy_model_factory.utils*), 17

M

mf() (*in module sqlalchemy_model_factory.pytest*), 18
mf_config() (*in module sqlalchemy_model_factory.pytest*), 18
mf_engine() (*in module sqlalchemy_model_factory.pytest*), 18
mf_registry() (*in module sqlalchemy_model_factory.pytest*), 18
mf_session() (*in module sqlalchemy_model_factory.pytest*), 18
module
    sqlalchemy_model_factory, 15
    sqlalchemy_model_factory.declarative, 15
    sqlalchemy_model_factory.pytest, 18
    sqlalchemy_model_factory.utils, 16

P

pytest (*class in sqlalchemy_model_factory.pytest*), 18

S

sqlalchemy_model_factory
    module, 15
sqlalchemy_model_factory.declarative
    module, 15
sqlalchemy_model_factory.pytest
    module, 18
sqlalchemy_model_factory.utils
    module, 16