Тестирование проектов, использующих SQLAlchemy

Перевезенцев Тимофей
@riffm
github.com/riffm

Речь идет о тестировании серверной части, т.е. python код

доклады про тестирование frontend'а идут в других секциях

Покрытие кода тестами

- 0% покрытие
- что-то между
- 100% покрытие

Kent Beck о покрытии

“I get paid for code that works, not for tests, so my philosophy is to test as little as possible to reach a given level of confidence…“

http://stackoverflow.com/questions/153234/how-deep-are-your-unit-tests/153565#153565

Интересные наблюдения

Сложный, большой проект работает в течении нескольких лет, только потому, что входящие данные валидны (данные приходят из другой системы). Это говорит о том, что точка зрения “тесты не нужны” имеет право на жизнь. Но если появляется надобность чуть-чуть где-то поменять, то разработчики “плюются”. Это говорит о том, что “тесты нужны” для данного проекта.

Интересные наблюдения

Если покрывать тестами чужой код, то получается непринужденный code review, что очень полезно. Поступая таким образом, разработчик получает навыки в обоих дисциплинах.

Что полезного даёт тестирование...

- рефакторинг
  - кода проекта
  - кода тестов (fixture...)
- новый код
  - есть хоть какая-то вероятность, что он не ломает старый
  - смело вносим новый код, если еще нет достаточно "стройного" решения, решение придет при рефакторинге

...а так же, что может случиться

- часто изменяемая предметная область
  - вы можете решать не ту проблему или делать это неправильно
- изменения неизбежны, хочется инструменты статического анализа, которые укажут на "мертвые тесты" (или код)
- если тесты к проекту пишут пару человек из десяти, то они обречены исправлять ошибки остальных, тех кто предпочитает “manual testing and automatic code review”

Этапы, которые проходит каждый тест

- установить начальное состояние
- произвести действие
- проверить корректность состояния или поведения
- почистить за собой

Этапы, которые проходит каждый тест

- установить начальное состояние
  - данные в БД, memcached/redis, Queue, файлы
  - может замедлять процесс тестирования
- произвести действие
  - например, сделать http запрос
- проверить корректность состояния или поведения
  - разные варианты утверждений, записи mock объектов
- почистить за собой
  - учитывая, что тест мог упасть
  - замедляет процесс тестирования

Начальное состояние БД

- много разных библиотек (fixture…)
  - навороченные, не всегда гибкие
  - дополнительный набор проблем, которые придется решать
- удобно использовать модели для описания начального состояния
  - это обычный код на python, который описывает начальное состояние в терминах вашей предметной области
  - надо понять как соблюсти принцип DRY

На помощь приходит python

```
class BookShopSample(Sample):
    def fawler(self):
        return Author(name='Martin Fawler')
    def refactoring(self):
        return Book(name='Refactoring', author=self.fawler)
    def patterns(self):
        return Book(name='Patterns', author=self.fawler)
```

Секреты класса Sample

- метакласс, который оборачивает публичные методы дескриптором, похожим на cached_property
- при обращении к атрибуту объекта Sample
  - будет вызван задекорированный метод
  - возвращаемое значение будет добавлено в сессию (sqlalchemy)
  - и закешировано

Плюсы подхода Sample

- гибкость
  - наследование
  - явное присвоение атрибута класса другому классу
  - т.о. можно использовать удобный для вас способ соблюдения DRY
- исчезает проблема с циклическими зависимостями в моделях

Храним авторов и книги отдельно

```
from samples import AuthorsSample

class BooksSample(AuthorsSample):
    def refactoring(self):
        return Book(name='Refactoring: Improving the Design '
                         'of Existing Code',
                    author=self.fawler)
```

Использование в тестах

```
from testutils import TestCase
from samples import BookShopSample

class BookShopTest(TestCase):
    def setUp(self):
        self.sample = BookShopSample(self.db)
        self.sample.create_all()
```

Очистка состояния

- перед каждым тестом пересоздать схему в базе
  - надежно
  - но очень, очень долго
  - во flask используют только данный способ очистки
- неявно оборачивать каждый тест в отдельную транзакцию и откатывать ее в конце
  - в django есть класс с таким подходом и еще один, который полностью отключает транзакции
  - pyramid
- получается либо очень медленно, либо хак

Давайте еще раз подумаем над этой проблемой

Controllable NIH is the way of my life...=)

Очистка состояния

- перед тестом у нас есть пустая база (с созданной схемой)
- в процессе установки начального состояния мы создаем новые записи
- в процессе тестирования мы создаем записи, удаляем и изменяем, существующие записи
- ?

Очистка состояния

- перед тестом у нас есть пустая база (с созданной схемой)
- в процессе установки начального состояния мы создаем новые записи
- в процессе тестирования мы создаем новые записи, удаляем и изменяем, существующие записи
- после тестирования удаляем существующие записи

Реализация очистки состояния

- используем сигнал after_flush сессии для получения сведений о новых записях
- сохраним ident каждой новой записи
- при очистке удаляем все записи, используя сохраненные ident'ы, если каких-либо записей в базе нет — нормально

Использование очистки состояния

```
from testalchemy import Restorable

with Restorable(session):
    #hack hack hack
```

Плюсы подхода

- работает быстро
  - с использованием ssd еще быстрее (советую)
- пересоздавать схему базы надо только при ее изменении

Большую часть работы делает SQLAlchemy

- IdentityMap
- UnitOfWork

DBHistory — история изменений

```
with DBHistory(session) as hist:
    resp = webcall(root.books.create.as_url, body=body)
book = hist.assert_created_one(Book)
assert_redirects(resp, root.books.item(id=book.id).as_url)
author = hist.assert_updated_one(Author)
assert author.books_count == session.query(Book)\
                                    .filter(author=author).count()
assert book.author == author
hist.assert_nothing_happened(except=(Book, Author))
```

github.com/riffm/testalchemy