Geoffrey Koh Follow

Apr 4, 2020 · 6 min read · Listen

Fun with Fixtures for Database Applications

Integration testing with databases is one of the most vital, yet commonly overlooked part of any software development process. This is, in part, due to the need to have a reachable database server active around somewhere. In this article, we explore unit and integration testing with databases that can be repeatedly spun up for testing, provides a uniform schema and test data, and can be easily torn down afterwards. All these can be done with the help of some nifty open source Python libraries that are readily available.

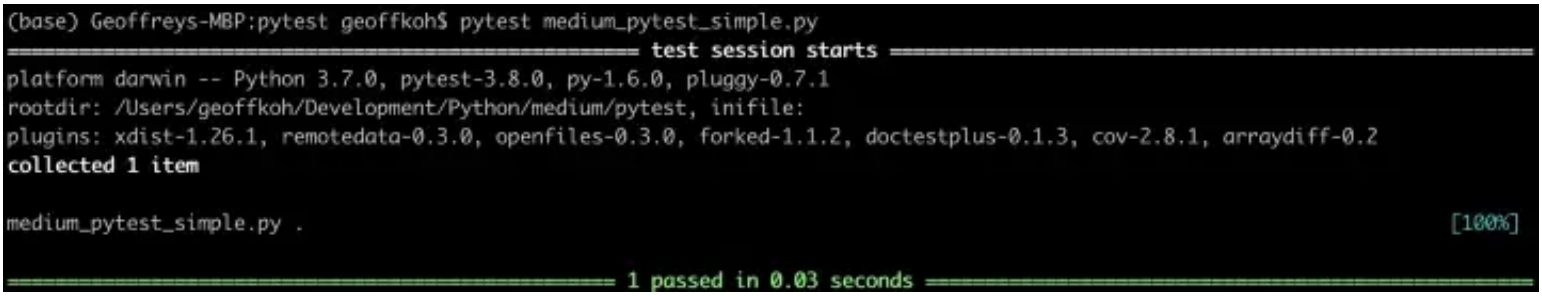
A Little Bit on Pytest

`pytest` is the current *de facto* testing framework for any Python application, surpassing in popularity to the `unittest` module, which comes as a standard package with all modern Python distributions. Vanilla use of `pytest` is very straight-forward. All one needs to do is to write a function with `test` as a substring of the function name, and run the `pytest` command on it.

```python
def add(a, b):
    """ Simple function to add two numbers """
    return a + b

def test_add():
    assert add(2, 4) == 6, 'The correct answer should be 6'
```

medium_pytest_simple.py hosted with ❤ by **GitHub** **view raw**



Example of testing a simple function with `pytest`

However, the real power of `pytest` comes when one understands all the bells and whistles underneath the hood, in particular **fixtures**, which can help perform various setting up and tearing down of relevant environments required to run test cases.

There are a lot of very good tutorials on `pytest` and thus we will not go into them in detail. This article focuses purely on using fixtures to set up temporary databases to achieve a more realistic and repeatable testing environment.

Fixtures

Fixtures are objects that set up certain conditions that are then used in testing. Think of them as functions that are called before running the actual test functions. In `pytest`, this can be done by marking a function with the decorator `@pytest.fixture`. Thereafter, you can use the fixture in any test function, simply by adding it as an argument to the function definition. A simple example is shown below.

```
# Standard imports
import requests
import sqlite3

# Third party imports
import pytest

@pytest.fixture
```

```
@pytest.fixture
def setup_database():
    """ Fixture to set up the in-memory database with test data """
    conn = sqlite3.connect(':memory:')
    cursor = conn.cursor()
    cursor.execute('''
        CREATE TABLE stocks
        (date text, trans text, symbol text, qty real, price real)''')
    yield conn


@pytest.fixture
def setup_test_data1(setup_database):
    cursor = setup_database
    sample_data = [
        ('2020-01-01', 'BUY', 'IBM', 1000, 45.0),
        ('2020-01-01', 'SELL', 'GOOG', 40, 123.0),
    ]
    cursor.executemany('INSERT INTO stocks VALUES(?, ?, ?, ?, ?)', sample_data)
    yield cursor


@pytest.fixture
def setup_test_data2(setup_database):
    cursor = setup_database
    sample_data = [
        ('2020-01-01', 'SELL', 'TESLA', 400, 233.0),
        ('2020-01-01', 'SELL', 'MSFT', 140, 980.0),
        ('2020-02-01', 'BUY', 'AMZN', 3000, 1200.0),
    ]
    cursor.executemany('INSERT INTO stocks VALUES(?, ?, ?, ?, ?)', sample_data)
    yield cursor


def test_with_sample_data2(setup_test_data2):
    # Test to make sure that there are 3 items in the database
    cursor = setup_test_data2
    assert list(cursor.execute('SELECT * FROM stocks')) == 3
```

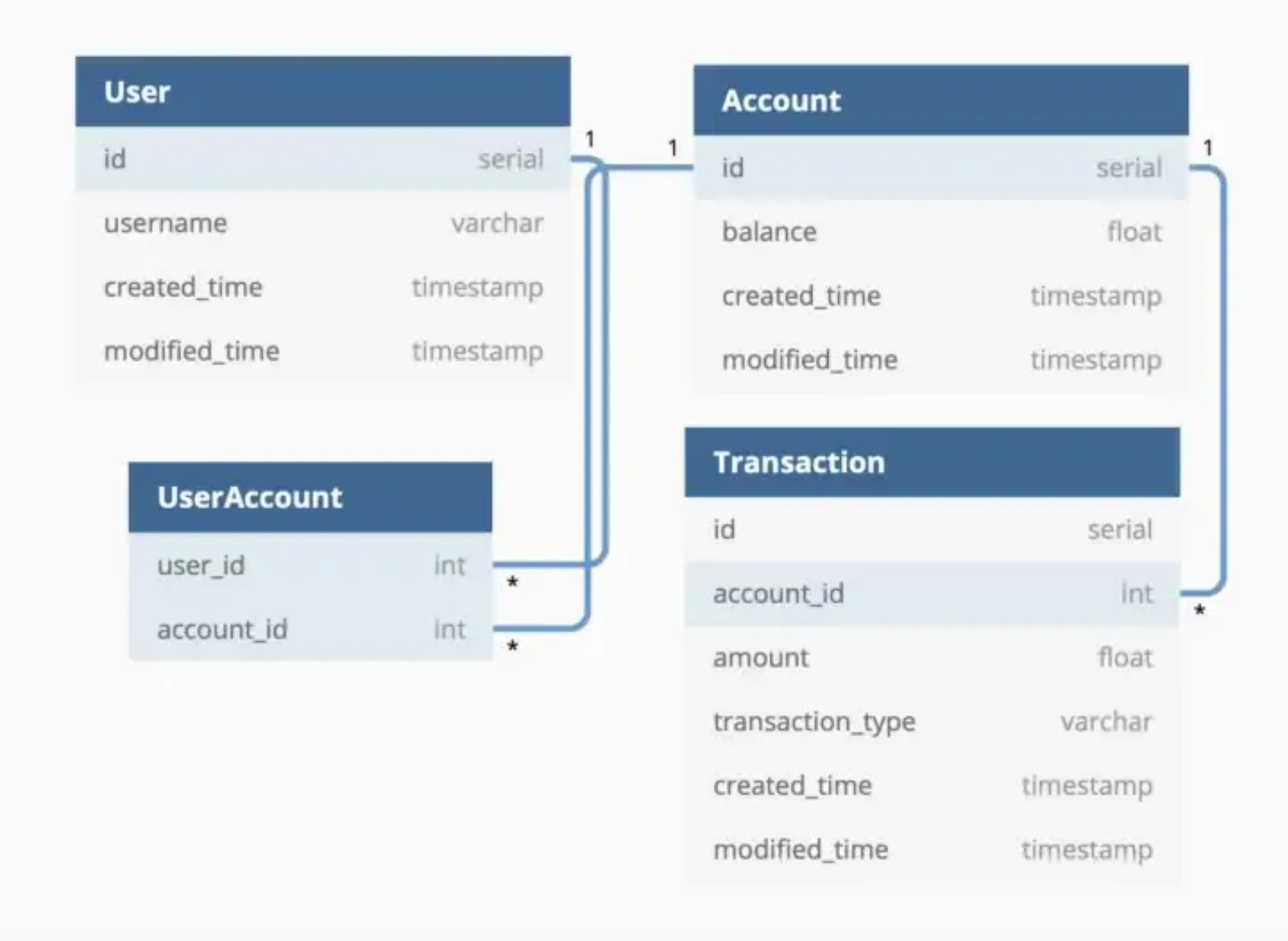
Using a fixture as the set up step for fixtures that fill in different test data

Later we will see some of these being used in the test codes for database connections.

Testing Database Interactions with Fixtures

We shall go through this tutorial with a simple example of an bank account transaction system (I mentioned in my previous article that I will not use examples that have no semblance to real life applications, but yet at the same time, simple enough to illustrate the concepts). As a proponent of **good clean design**, I will also recommend using `SQLAlchemy` (https://www.sqlalchemy.org) to model the various business objects.

In this system, each customer will be a **user**, and each user can have one or more **accounts**. At the same time, an **account** can belong to a single user, or it can be a joint account belonging to **multiple users**. Every time the user credits or debits into the account, a **transaction** will be created. The Entity-Relation diagram reflecting the relationship is shown below.



ER Diagram for the simple bank account application

Now that we have the business objects, their Object Relation Model definitions in SQLAlchemy is then shown in the snippet below.

```
# Third party imports
from sqlalchemy import Column, Integer, String, DateTime, Float, ForeignKey
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship
from sqlalchemy.sql import func

Base = declarative_base()


class User(Base):
    """ Model class to represent a user """

    __tablename__ = 'user'

    id = Column(Integer, primary_key=True, autoincrement=True)
    username = Column(String, nullable=False, unique=True)
```

```
    created_time = Column(DateTime(timezone=True),
                          server_default=func.now())
    modified_time = Column(DateTime(timezone=True),
                           server_default=func.now(),
                           onupdate=func.now())

```

```
# Standard imports
from typing import List

# Third party imports
from sqlalchemy.orm.session import Session
from sqlalchemy.orm.exc import NoResultFound

# Application imports
from models import Account, User, Transaction


def get_user(username: str, session: Session) -> User:
    """ Gets the user by username """

    try:
        user = session.query(User).filter(User.username == username).one()
        return user
    except NoResultFound:
        return None

# end get_user()


def get_accounts_by_user(username: str, session: Session) -> List[Account]:
    """ Retrieves the accounts given the username """

    user = get_user(username, session)
    accounts = list(session.query(Account).filter(
        Account.users.contains(user)).all())
    return accounts

# end get_accounts_by_user


def compute_balance(username: str, session: Session) -> float:
    """ Computes the balance based on the username """

    accounts = get_accounts_by_user(username, session)
    balance = sum([account.balance for account in accounts])
    return balance

# end compute_balance()
```

```
43
44
45  def debit(account_id: int, amount: float, session: Session) -> float:
46      """ Debits amount into the account """
47
48      try:
49          account = session.query(Account).filter(Account.id == account_id).one()
50          account.balance -= amount
51          transaction = Transaction(account_id=account_id,
52                                    amount=amount,
53                                    transaction_type='DEBIT')
54          session.add(transaction)
55          session.commit()
56          return account.balance
57      except NoResultFound:
58          return 0
59
60  # end debit()
```

```
 1  # Third party imports
 2  import pytest
 3  from sqlalchemy import create_engine
 4  from sqlalchemy.orm import sessionmaker
 5
 6  # Application imports
 7  from core import get_user, get_accounts_by_user, compute_balance, debit
 8  from models import Base, User, Account, UserAccount, Transaction
 9
10
11  @pytest.fixture(scope='function')
12  def setup_database():
13
14      engine = create_engine('sqlite://')
15      Base.metadata.create_all(engine)
16      Session = sessionmaker(bind=engine)
17      session = Session()
18      yield session
19      session.close()
20
21  # end setup_database()
22
23
24  @pytest.fixture(scope='function')
25  def dataset(setup_database):
26
27      session = setup_database
28
29      # Creates user
```

```
30      john = User(username='john')
31      mary = User(username='mary')
32      session.add(john)
```

```
(base) Geoffreys-MBP:pytest geoffkoh$ pytest -sv medium_pytest_functions.py
============================= test session starts ==============================
platform darwin -- Python 3.7.0, pytest-3.8.0, py-1.6.0, pluggy-0.7.1 -- /anaconda3/bin/python
cachedir: .pytest_cache
rootdir: /Users/geoffkoh/Development/Python/medium/pytest, inifile:
plugins: xdist-1.26.1, remotedata-0.3.0, openfiles-0.3.0, forked-1.1.2, doctestplus-0.1.3, cov-2.8.1, arraydiff-0.2
collected 1 item

medium_pytest_functions.py::test_database PASSED
```

```
39      joint_account = Account(balance=20.0)
40      john.accounts.append(john_account)
41      mary.accounts.append(mary_account)
42      john.accounts.append(joint_account)
43      mary.accounts.append(joint_account)
44      session.add(john_account)
45      session.add(mary_account)
46      session.add(joint_account)
47      session.commit()
48
49      yield session
50
51  # end dataset_1
52
53
54  def test_database(dataset):
55
56      # Gets the session from the fixture
57      session = dataset
58
59      # Do some basic checking
60      assert len(session.query(User).all()) == 2
61      assert len(session.query(Account).all()) == 3
62      assert len(session.query(UserAccount).all()) == 4
63
64      # Retrieves John and Mary
65      john = get_user('john', session)
66      mary = get_user('mary', session)
67
68      # Checks their accounts
69      assert len(get_accounts_by_user(john.username, session)) == 2
70      assert len(get_accounts_by_user(mary.username, session)) == 2
71
72      # Checks the balance
73      assert compute_balance(john.username, session) == 30.0
74      assert compute_balance(mary.username, session) == 25.0
```

```
1   # Standard imports
2   import tempfile
3
```

```python
# Third party imports
import pytest
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from pytest_postgresql import factories


# Using the factory to create a postgresql instance
socket_dir = tempfile.TemporaryDirectory()
postgresql_my_proc = factories.postgresql_proc(
    port=None, unixsocketdir=socket_dir.name)
postgresql_my = factories.postgresql('postgresql_my_proc')


@pytest.fixture(scope='function')
def setup_database(postgresql_my):

    def dbcreator():
        return postgresql_my.cursor().connection

    engine = create_engine('postgresql+psycopg2://', creator=dbcreator)
    Base.metadata.create_all(engine)
    Session = sessionmaker(bind=engine)
    session = Session()
    yield session
    session.close()

# end setup_database()
```

medium_pytest_functions_pgsql.py hosted with ❤ by **GitHub** **view raw**

- `pytest-redis` — https://pypi.org/project/pytest-redis/
- `pytest-consul` — https://pypi.org/project/pytest-consul/
- `pytest-kafka` — https://pypi.org/project/pytest-kafka/
- … And more

Coincidentally, they are all created with similar interfaces, such that the process of setting them up goes through roughly the same processes.

Hopefully this article can give users a clearer understanding of how to better test database-driven applications, and thereby leading to more robust and reliable software.

Given the ubiquity of Python and `pytest`, if you find that a plugin for a database you want to test for is missing, you can either write one (or contact me), or appeal to the general Python