looking for an IT consultant?
aws
Creating of a chatbot
Integration with payment providers
Setting up a CI/CD pipeline
Help with Serverless stack
Integration with messaging services
Building Web scrapers or automation
Custom development using Python
Python or cloud coaching

September 6, 2020

#Python | #pytest | #blog

Testing database with pytest

Other pytest articles:

Why testing is important

Types of tests

Test driven Development

Hello, World!

Selecting tests with pytest

Testing HTTP client with pytest

Testing database with pytest

Advanced fixtures with pytest

Pytest plugins

We are going to use a database in our number testing application as a cache for API call results - API calls can be costly and we don’t want to check the same number twice against it.

Let’s think about what we want from the database caching service

- We should be able to query database to get the validity of the number if it's present
- We should be able to save number status that we got from API to database
- Also we want to generate a report - a percentage of valid numbers in the database

Now we start with writing tests for these user stories, but first let's think about the design of our service.

Monkey Patching and responses library that we used in the previous unit are python and pytest test specific features, but there is another one that's used in other programming languages. This approach utilizes a common design pattern called dependency injection.

I propose for this service to be represented as a class. The constructor of the class will accept a DB session so we implement dependency injection that will allow testing the class easily

All test cases will be using a fixture - session. The first case will test the get method of the class in case of an existing number The second case - same get method but for the number that is not in the database - we expect to receive None

Copy

```
def test_get(session): # 1
    cache = CacheService(session) # 2
    existing = cache.get_status('+3155512345') # 3
    assert existing

def test_get_unknown(session):
    cache = CacheService(session)
    assert cache.get_status('+315554444') is None
```

1. Use a fixture `session` for test cases - see implementation later
2. Pass `session` instance to our Cache class constructor
3. Invoke `get_status` of the class, remember it'd TDD. So first tests, then real code

Next test will test the save method, and will utilize the get method again to check if it was saved.

Copy

```
def test_save(session):
    number = '+3155512346'
    cache = CacheService(session)
    cache.save_status(number, True)
    existing = cache.get_status(number)
    assert existing
```

The last case will generate the report - which is just a ratio of valid numbers in the database. We use a save method to seed tables with data.

Copy

```
def test_report(session):
    cache = CacheService(session)
    cache.save_status('+3155512346', True)
    cache.save_status('+3155512347', False)
    cache.save_status('+3155512348', False)
    ratio = cache.generate_report()
    assert ratio == 0.5
```

Of course there is no such fixture as session now and there is no database and tables. I'm going to use an in-memory sqlite database and create a table. Now this fixture will be invoked before every time when we pass it as an argument to the test case.

But the session should be closed afterwards - for that we can separate fixture code in 2 parts that are separated by the yield keyword. First part if executed before the test case, the second - afterwards as a cleanup. And we are going to yield a session from the fixture for it to be used in test case code.

Copy

```
@pytest.fixture # 1
def session():
    connection = sqlite3.connect(':memory:') # 2
    db_session = connection.cursor()
    db_session.execute('''CREATE TABLE numbers
                          (number text, existing boolean)''') # 3
    db_session.execute('INSERT INTO numbers VALUES ("+3155512345", 1)') # 4
    connection.commit()
    yield db_session # 5
    connection.close()
```

1. Create a new fixture
2. Use sqlite3 from standard library and connect to in memory database
3. Use DB cursor to create a table
4. Put some data in the table
5. `yield` splits our fixture in 2 parts

Now there are 2 things that the fixture does - it creates a session and sets up the database. I'm going to extract the second part into a separate fixture. It will accept a session as a parameter.

Copy

```
@pytest.fixture
def session(): # 1
    connection = sqlite3.connect(':memory:')
    db_session = connection.cursor()
    yield db_session
    connection.close()


@pytest.fixture
def setup_db(session): # 2
    session.execute('''CREATE TABLE numbers
                          (number text, existing boolean)''')
    session.execute('INSERT INTO numbers VALUES ("+3155512345", 1)')
    session.connection.commit()
```

1. Remove all setup code from fixture `session`
2. Use session fixture in another fixture `setup_db`, where all setup code is moved

Now to use it in the test I'm going to decorate test case with use fixture instead of passing setup_db as a parameter - we don't need this fixture in the test case code - we need this fixture only to be executed

Copy

```
@pytest.mark.usefixtures("setup_db") # 1
def test_get(session):
    ...
```

1. Fixture is invoked automatically

Also it looks like we instantiate caching service in every test case - there is a lot of duplicate code. Again we can create a fixture - caching service and pass it to the test cases instead of session.

Copy

```
@pytest.fixture
def cache(session): # 1
    return CacheService(session)

@pytest.mark.usefixtures("setup_db")
def test_get(cache): # 2
    existing = cache.get_status('+3155512345')
    assert existing
```

1. Create a more high level fixture that represents our mock in memory Cache
2. Use newly created fixture in test case code

Tests and fixtures are covered - it’s time to write actual code. In the test I’ve made a design decision to make it a class with a session injected. Also I decided to use sqlite and it’s driver from the standard python library. The code of the actual cache service is pretty simple in that case.

Copy

```
class CacheService:
    def __init__(self, session): # 1
        self.session = session # 2

    def get_status(self, number):
        self.session.execute('SELECT existing FROM numbers WHERE number=?', (number,))
        return self.session.fetchone()

    def save_status(self, number, existing):
        self.session.execute('INSERT INTO numbers VALUES (?, ?)', (number, existing))
        self.session.connection.commit()

    def generate_report(self):
        self.session.execute('SELECT COUNT(*) FROM numbers')
        count = self.session.fetchone()
        self.session.execute('SELECT COUNT(*) FROM numbers WHERE existing=1')
        count_existing = self.session.fetchone()
        return count_existing[0]/count[0]
```

1. Class constructor accepts a argument - `session`. Its instance should be created outside and have a certain mathods that are used in this class. Due to duck typing it doesn’t matter if it’s fake object or a real session
2. Store session in class attributes

But since our Caching service accepts a session in its constructor - we can inject a mock object and validate how our code calls the database. And we can rewrite one of the first test cases for a get method like so

Copy

```
from unittest.mock import MagicMock

def test_get_mock():
    session = MagicMock() # 1
    executor = MagicMock()
    session.execute = executor
    cache = CacheService(session) # 2
    cache.get_status('+3155512345')
    executor.assert_called_once_with('SELECT existing FROM numbers WHERE number=?', ('
```

1. Mock object - it can have any methods
2. That mock we will pass to Cache
3. To check what is actually called we use different types of asserts

There are some important differences when using mocks.

1. They are in memory abstract objects
2. We need to manually define methods, like here for a session mock we define a method execute
3. They have special types of assertions - here we don't check the data like in previous examples, but the behavior.

There is a number of different assert methods available for mock. In this example, I'm checking that our caching component constructed the query properly and uses bind variables to avoid SQL injection.

Unittest.mock is a powerful library - it's docs are available at https://docs.python.org/3/library/unittest.mock.html. And there is a pytest specific wrapper that can be found here https://pypi.org/project/pytest-mock/

In this unit you've learned a bit more about mocking. In the next one you'll get familiar with more advanced usages of pytest fixtures.

Similar articles:

- Advanced fixtures with pytest
- Hello, World!
- Pytest plugins
- Selecting tests with pytest
- Test driven Development

© Alexey Smirnov 2023