Практическая работа №8. Тестирование сервисов

Цель работы

Основной целью данной работы является изучение видов pre-deployment тестирования и получение практических навыков написания автотестов и их внедрения в CI/CD конвейеры.

Теоретическое введение

При разработке монолитного приложения одной из проблем при увеличении размеров сервиса становится его тестирование. При внесении любого изменения в работу приложения необходимо производить тестирование всего функционала, так как из-за высокой связности модулей изменения, внесённые в одной части системы, могут сломать работу приложения в совершенно другом месте.

Благодаря распределённой природе систем с микросервисной архитектурой, необходимость прогонять все существующие тесты пропадает. Каждый сервис имеет свой собственный набор тестов, которые выполняются в автоматическом режиме при каждой загрузке кода в репозиторий. Это происходит за счёт встраивания тестирования в конвейер непрерывной интеграции.

В процессе выполнения CI/CD возможно выполнение только pre-deployment тестов. Как правило такие тесты предполагают полную изоляцию тестируемых сервисов от других систем. К pre-deployment тестированию относятся unit-тесты, интеграционные тесты, контрактные тесты и компонентные тесты (e2e уровня одного сервиса). Работа с контрактными тестами в данной работе не рассматривается.

Unit-тесты

Unit-тестирование предназначено для проверки работоспособности отдельно взятых функций сервиса. Как правило такие тесты пишутся на том

же языке, что и сам сервис, чтобы было проще вызывать отдельные функции из кода.

Предполагается, что таких тестов в проекте должно быть больше всего. Unit-тесты проверяют, корректное выполнение конкретных функций. От качества тестирования данного типа зависит правильность работы системы в целом.

При проведении unit-тестирования недопустима работа со внешними сервисами, так как все входные и выходные параметры должны быть чётко определены. Внешняя зависимость вносит неопределённость за счёт того, что её поведение может меняться.

На этапе unit-тестирования необходимо тестировать самые разные объекты сервиса. Начать стоит с тестирования бизнес-объектов. В процессе работы над бизнес-логикой сервиса создавались объекты, содержащие в себе какие-то правила или поведение. Например, обязательные и необязательные поля, различные запрещённые значения, разнообразные конструкторы. Всё это стоит тестировать.

```python
# /tests/unit/test_delivery_model.py

import pytest
from uuid import uuid4
from datetime import datetime
from pydantic import ValidationError

from app.models.deliveryman import Deliveryman
from app.models.delivery import Delivery, DeliveryStatuses


@pytest.fixture()
def any_deliveryman() -> Deliveryman:
    return Deliveryman(id=uuid4(), name='delliveryman')


def test_delivery_creation(any_deliveryman: Deliveryman):
    id = uuid4()
    address = 'address'
    date = datetime.now()
    status = DeliveryStatuses.DONE
    delivery = Delivery(id=id, address=address, date=date,
                        status=status, deliveryman=any_deliveryman)
```

```
    assert dict(delivery) == {'id': id, 'address': address, 'status': status,
                              'deliveryman': any_deliveryman, 'date': date}


def test_delivery_address_required(any_deliveryman: Deliveryman):
    with pytest.raises(ValidationError):
        Delivery(id=uuid4(), date=datetime.now(),
                 status=DeliveryStatuses.ACTIVATED, deliveryman=any_deliveryman)


def test_delivery_date_required(any_deliveryman: Deliveryman):
    with pytest.raises(ValidationError):
        Delivery(id=uuid4(), address='str',
                 status=DeliveryStatuses.ACTIVATED, deliveryman=any_deliveryman)


def test_delivery_status_required(any_deliveryman: Deliveryman):
    with pytest.raises(ValidationError):
        Delivery(id=uuid4(), date=datetime.now(),
                 address='str', deliveryman=any_deliveryman)
```

Кроме тестирования бизнес-объектов здесь также стоит протестировать разные функции, которые используются в коде. На этом этапе важно помнить про то, что работа с внешними зависимостями недопустима. Ко внешним зависимостям относятся и другие сервисы, и базы данных, и сервисы, лежащие за пределами разрабатываемой системы. Принято такие системы подменять моковыми, которые лишь имитируют поведение реальных. Именно на этом этапе большую роль сыграет использование паттерна Dependency injection, который позволяет достаточно просто внедрять использование моковых сервисов в код. В примере ниже приводится часть тестов сервиса управления доставками. В конструктор сервиса передаётся локальный репозиторий доставок вместо того, который работает с БД. Оба репозитория реализуют один интерфейс и тестируются на одних и тех же тестах, но на разных этапах: локальный – unit-тесты, с БД – интеграционные тесты. Здесь также важно протестировать все исключения.

```
# /tests/unit/test_delivery_service.py

import pytest
from uuid import uuid4, UUID
```

```python
from datetime import datetime

from app.services.delivery_service import DeliveryService
from app.models.delivery import DeliveryStatuses
from app.repositories.local_delivery_repo import DeliveryRepo
from app.repositories.local_deliveryman_repo import DeliverymenRepo


@pytest.fixture(scope='session')
def delivery_service() -> DeliveryService:
    return DeliveryService(DeliveryRepo(clear=True))


@pytest.fixture()
def deliveryman_repo() -> DeliverymenRepo:
    return DeliverymenRepo()


@pytest.fixture(scope='session')
def first_delivery_data() -> tuple[UUID, str, datetime]:
    return (uuid4(), 'address_1', datetime.now())


@pytest.fixture(scope='session')
def second_delivery_data() -> tuple[UUID, str, datetime]:
    return (uuid4(), 'address_2', datetime.now())


def test_empty_deliveries(delivery_service: DeliveryService) -> None:
    assert delivery_service.get_deliveries() == []


def test_create_first_delivery(
    first_delivery_data: tuple[UUID, str, datetime],
    delivery_service: DeliveryService
) -> None:
    order_id, address, date = first_delivery_data
    delivery = delivery_service.create_delivery(order_id, date, address)
    assert delivery.id == order_id
    assert delivery.address == address
    assert delivery.date == date
    assert delivery.status == DeliveryStatuses.CREATED
    assert delivery.deliveryman == None


def test_create_first_delivery_repeat(
    first_delivery_data: tuple[UUID, str, datetime],
    delivery_service: DeliveryService
) -> None:
    order_id, address, date = first_delivery_data
    with pytest.raises(KeyError):
        delivery_service.create_delivery(order_id, date, address)
```

После написания unit-тестов и проверки их работоспособности необходимо добавить их выполнение в CI/CD конвейер.

```yaml
# /.github/workflows/ci-cd.yml

...
  build-and-push-to-yc:
    name: Build and push to YandexCloud Registry
    runs-on: ubuntu-latest
    steps:
      ...
      - name: Prepare unit tests
        run: |
          sudo apt update
          sudo apt install -y python3-pip
          pip install -r requirements.txt

      - name: Run unit tests
        run: |
          pytest ./tests/unit
...
```

Интеграционные тесты

Данный тип тестирования предназначен для проверки подключений разрабатываемого сервиса к другим системам, например к базе данных. Для такого типа тестирования необходимо специально запускать копии систем, подключение к которым планируется тестировать.

В случае с сервисом управления доставкой в интеграционное тестирование уйдёт весь репозиторий заказов, работающий с БД.

```python
# /tests/integration/test_delivery_repo_db.py

import pytest
from uuid import uuid4
from time import sleep
from datetime import datetime
from app.models.delivery import Delivery, DeliveryStatuses
from app.repositories.db_delivery_repo import DeliveryRepo
from app.repositories.local_deliveryman_repo import DeliverymenRepo

@pytest.fixture()
def delivery_repo() -> DeliverymenRepo:
    repo = DeliveryRepo()
```

```
    return repo

@pytest.fixture(scope='session')
def deliveryman_repo() -> DeliverymenRepo:
    return DeliverymenRepo()

@pytest.fixture(scope='session')
def first_delivery() -> Delivery:
    return Delivery(id=uuid4(), address='address', date=datetime.now(),
status=DeliveryStatuses.CREATED)

@pytest.fixture(scope='session')
def second_delivery() -> Delivery:
    return Delivery(id=uuid4(), address='address1', date=datetime.now(),
status=DeliveryStatuses.CREATED)

def test_empty_list(delivery_repo: DeliverymenRepo) -> None:
    assert delivery_repo.get_deliveries() == []

def test_add_first_delivery(first_delivery: Delivery, delivery_repo:
DeliverymenRepo) -> None:
    assert delivery_repo.create_delivery(first_delivery) == first_delivery

def test_add_first_delivery_repeat(first_delivery: Delivery, delivery_repo:
DeliverymenRepo) -> None:
    with pytest.raises(KeyError):
        delivery_repo.create_delivery(first_delivery)

def test_get_delivery_by_id(first_delivery: Delivery, delivery_repo:
DeliverymenRepo) -> None:
    assert delivery_repo.get_delivery_by_id(
        first_delivery.id) == first_delivery

def test_get_delivery_by_id_error(delivery_repo: DeliverymenRepo) -> None:
    with pytest.raises(KeyError):
        delivery_repo.get_delivery_by_id(uuid4())
def test_add_second_delivery(first_delivery: Delivery, second_delivery: Delivery,
delivery_repo: DeliverymenRepo) -> None:
    assert delivery_repo.create_delivery(second_delivery) == second_delivery
    deliveries = delivery_repo.get_deliveries()
    assert len(deliveries) == 2
    assert deliveries[0] == first_delivery
    assert deliveries[1] == second_delivery
...
```

После написания интеграционных тестов и проверки их работоспособности необходимо добавить их выполнение в CI/CD конвейер.

```yaml
# /.github/workflows/ci-cd.yml

name: CI/CD pipline

...
  build-and-push-to-yc:
    name: Build and push to YandexCloud Registry
    runs-on: ubuntu-latest
    steps:
      ...
      - name: Prepare integration tests
        run: |
          sudo docker run -e POSTGRES_PASSWORD=password -p 5432:5432 -d
postgres:14
          rm ./.env
          echo "AMQP_URL=$AMQP_URL
          POSTGRES_URL=$POSTGRES_URL
          " > .env
          sleep 5
          alembic upgrade head
        env:
          AMQP_URL: amqp://guest:guest123@51.250.26.59:5672/
          POSTGRES_URL: postgresql://postgres:password@localhost:5432/postgres

      - name: Run integration tests
        run: |
          pytest ./tests/integration
...
```

Компонентные тесты

В качестве компонентных тестов в микросервисной архитектуре можно рассматривать end-to-end (e2e) тесты применительно к единственному сервису. На данном этапе происходит тестирование сервиса не изнутри, а снаружи. Больше не вызываются внутренние функции, теперь необходимо выполнять запросы к сервису через определённые в нём интерфейсы. В случае с сервисом управления доставкой одним из таких интерфейсов является описанный внутри HTTP-роутер.

```python
# /tests/e2e/test_delivery_router.py

import time
```

```
import pytest
import requests
from uuid import UUID, uuid4
from datetime import datetime

from app.models.delivery import Delivery, DeliveryStatuses


time.sleep(5)
base_url = 'http://localhost:8000/api'


@pytest.fixture(scope='session')
def first_delivery_data() -> tuple[UUID, str, datetime]:
    return (uuid4(), 'address_1', datetime.now())


@pytest.fixture(scope='session')
def second_delivery_data() -> tuple[UUID, str, datetime]:
    return (uuid4(), 'address_2', datetime.now())


def test_get_deliveries_empty() -> None:
    assert requests.get(f'{base_url}/delivery').json() == []


def test_add_delivery_first_success(
    first_delivery_data: tuple[UUID, str, datetime]
) -> None:
    order_id, address, date = first_delivery_data
    delivery = Delivery.model_validate(requests.post(f'{base_url}/delivery',
json={
        'order_id': order_id.hex,
        'date': str(date),
        'address': address
    }).json())
    assert delivery.id == order_id
    assert delivery.status == DeliveryStatuses.CREATED
    assert delivery.date == date
    assert delivery.address == address
...
```

После написания компонентных тестов и проверки их работоспособности необходимо добавить их выполнение в CI/CD конвейер.

```
# /.github/workflows/ci-cd.yml

...
jobs:
  test:
    name: Run tests
```

```
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      ...

      - name: Prepare e2e tests
        run: |
          sudo docker stop pg-tests
          sudo docker build . --file Dockerfile --tag back-tests
          sudo docker run -e POSTGRES_PASSWORD=password -p 5432:5432 -d
postgres:14
          sleep 5
          alembic upgrade head
          sudo docker run -p 8000:80 -e
AMQP_URL=amqp://guest:guest123@51.250.26.59:5672/ -e
POSTGRES_URL=postgresql://postgres:password@host.docker.internal:5432/postgres -d
back-tests


      - name: Run e2e tests
        run: |
          pytest ./tests/integration
```

Все тесты, описанные в CI/CD конвейере, лучше всего вынести в отдельную работу, которая будет запускаться перед остальными.

Ссылка на проект: https://github.com/IvLaptev/MA_6.

Задание на самостоятельную работу

1. Опираясь на результаты Практических работ 6 и 7 необходимо выбрать два вида тестирования из трёх (unit, интеграционное, компонентное) и покрыть разработанные сервисы тестами.
2. После разработки тестов, добавить их выполнение в CI/CD конвейеры.