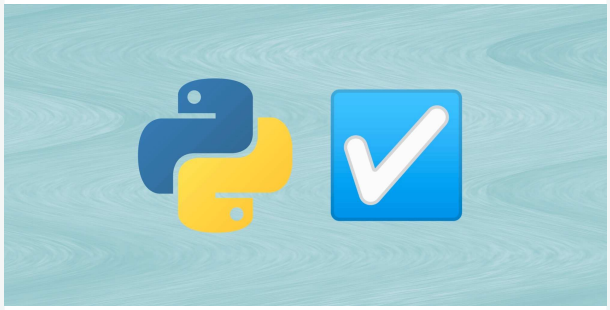
Алена Вахтина 15 июня 2021

🐍 Python для автоматизации тестирования: создаем несложный REST-тест за 4 шага

Python для инженера по Quality Assurance – универсальный «швейцарский нож», которым легко воспользоваться. Рассказываем, как создать автотест за 4 простых шага.



Python в тестирование, сферы его применения

Автоматизированное тестирование давно стало обязательным в сфере IT. Предъявляемые к инструментам инженера по Quality Assurance (QA) требования все выше, а выбор средств все больше: сейчас для улучшения качества продукта можно использовать даже машинное обучение. Python можно назвать швейцарским ножом в сфере тестирования. Нужно написать UI-тесты? Используйте Python. Требуется сгенерировать большое количество данных? Снова Python. Хотите создать бота для тестирования WoT? Тоже Python. Удобнее всего писать на Python тесты REST API.

План REST-теста

Чтобы написать хороший REST-тест нужно сделать следующее:

- сгенерировать данные;
- положить их в базу данных;
- отправить REST-запросы;
- сверить результаты с ожидаемыми;
- сгенерировать отчет по результатам.

Для каждого действия есть идеально подходящая библиотека Python.

1. Создание тестовых данных

Можно сгенерить рандомные данные или взять их из файла csv. Например, для создания нового пользователя нужно заполнить поле ФИО, дату рождения или возраст, почту и пароль (почта будет служить логином):

```
import random
import string
import datetime
# генерация случайного числа
age1 = random.randrange(15))
# генерация числа случайного в промежутке от 1 до 100 с шагом 3
age2 = random.randrange(0, 101, 3))
# генерация числа с плавающей точкой в промежутке от 5.2 до 7.9
print(random.uniform(5.2, 7.9))
# генерация строки из 10 случайных символов
letters = string.ascii_letters
password = ''.join(random.choice(letters) for i in range(10))
# выбор случайного значения из list
name = random.choice(['Oliver', 'William', 'James'])
mail = name+'@'+random.choice(['mail.ru', 'gmail.com', 'ya.ru'])
```

Те же самые данные можно получить из заранее подготовленного файла в формате csv:

```
import csv
with open('user.csv') as csv_file:
   csv_reader = csv.reader(csv_file, delimiter=',')
   for row in csv_reader:
        age = row['age']
        # и так далее...
        name = row['name']
```

2. Добавление тестовых данных в базу

Информацию можно добавить в базу данных, что будет быстрее использования REST-запросов: Python умеет работать как реляционными, так и нереляционными СУБД. Рассмотрим отрывок кода, в котором происходит подключение к БД и добавление нового пользователя. В примере используется PostgreSQL, потому что это один из наиболее распространенных вариантов.

```
import psycopg2
import logging

# аргументы для подключения к БД
```

```
def conDB():
    try:
        conn = psycopg2.connect(DB_ARGS)
    except:
        logging.error("Unable to connect to the database.")
        return None
    cur = conn.cursor(cursor_factory=psycopg2.extras.DictCursor)
    return cur
```

***Примечание:** код будет работать, если локально поднята БД `test`, и в ней есть таблица `user` с колонками `name`, `age`, `mail`, `pass`.*

Более подробная информация доступна на сайте.

3. Первые REST-запросы

Библиотек для работы с с REST-запросами существует великое множество. Мне больше всего нравятся **aiohttp** и **requests**. Для написания тестов удобнее requests. С помощью POST-запроса создадим нового пользователя и после этого GET-запросом проверим, что он действительно был добавлен.

```
import requests

# тело запроса для создания нового пользователя
user = {"name": "Fred", "age": 25,"mail":"fr@mal.com", "password": "134513
r = requests.post("http://localhost/users/", data=user)
# напечатать код запроса
print(r.status_code)
# GET запрос на получение пользователя по id
url = "http://localhost/users/" + str(r.json()['id'])
r = requests.get("http://localhost/users/", data=user)
print(r.text)
```

***Примечание:** код будет работать, если локально поднят сервис, принимающий запросы **POST** и **GET** на создание пользователя и получение информации о нем.*

4. Использование библиотеки PyTest

*Приведенный выше код еще не является полноценным тестом. Если оставить его в таком виде, то будет сложно поддерживать. Для систематизации, а также для улучшения качества можно использовать библиотеку **pyTest**.*

PyTest – это удобный инструмент, который автоматически находит написанные тесты, запускает тесты и пишет отчеты с результатом. Библиотека активно развивается и поддерживается. Как использовать все ее функции можно почитать в книге “Python Testing with pytest” Брайана Оккена.

Разделим предыдущий код на 2 полноценных теста: создание пользователя и получение информации о нем по id.

```
import pytest
import requests
import json

# тест на создание пользователя и проверку успешного создания
def test create_user():
    user = {"name": "Fred", "age": 25, "mail":"fr@mal.com", "password": "1
    url = "http://localhost/users/"
    r = requests.post(url, data=user)
    try:
        r.raise_for_status()
    except requests.exceptions.HTTPError as e:
        print('ERROR: %s' % e)
    assert r.text == "Ok"
```

***Примечание:** код будет работать, если локально поднят сервис, принимающий запросы **POST** и **GET** на создание пользователя и получение информации о нем.*

В втором тесте на получение пользователя был использован стандартный модуль **json**. Он значительно упрощает работу с JSON-объектами, позволяя не задумываться над сериализацией и десериализацией, обращением по ключу и поиске значения. Подробнее почитать об этом можно, например, здесь.

Пара слов об отчетах

Библиотека pyTest уже умеет генерировать отчеты. Если хочется получить более подробные отчеты, содержащие информацию о покрытие кода, красивые графики и схемы, то можно дополнительно подключить библиотеку **allure**. Не буду дополнительно расписывать ее, поскольку это уже факультативно подключаемая библиотека. Для основной работы будет хватать и pyTest. Почитать об allure можно тут.

Плюсы Python

Тесты готовы. Отчеты получены. На каждое действия для Python есть удобная в использовании и легкая в освоении библиотека. На этом плюсы популярного языка программирования не заканчиваются. Python быстро развивается, постоянно появляются новые библиотеки для инженеров по QA, комьюнити растет, и всегда найдутся люди, готовые ответить вам на вопрос и помочь решить проблему. Также Python отличается одним из самых низких порогов вхождения, поэтому вам будет удобно начинать путь от ручного тестирования к автоматизации и покрытию продукта REST-тестами.

Минусы Python в автотестах

Конечно у любого инструмента есть и минусы. При разрастание проекта количество тестов обычно увеличивается в геометрической прогрессии. Тестирование будет идти слишком долго, что начнет замедлять процесс работы всей команды. Python – не самый быстрый язык. В какой-то момент придется запускать тесты параллельно, что достаточно сложно. Работать с многопоточностью в Python сложно, но когда такая проблема возникает перед тестировщиком, он уже имеет достаточно навыков и опыта для ее решения.

Выводы