GeekBrains

Автоматизация тестирования веб-приложений на Python

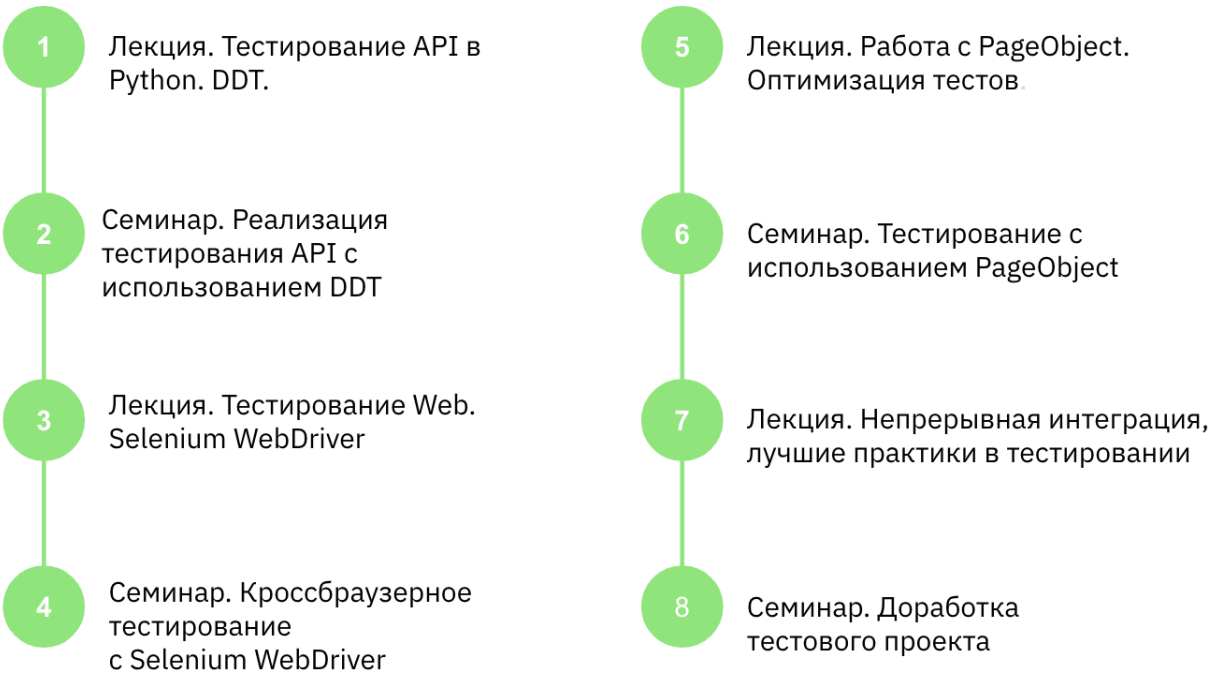
Лекция 1
Тестирование API в Python. DDT

Оглавление

Введение

Добро пожаловать на курс «Автоматизация тестирования веб-приложений на Python»!

Курс состоит из 4 лекций и 4 семинаров, на которых мы научимся решать задачи автоматизации тестирования веб-приложений с использованием языка Python.



Освоив этот курс, вы на реальных примерах научитесь решать основные задачи в области автоматизации тестирования веб-приложений с использованием pytest.

На этом уроке мы узнаем:

- Особенности тестирования API, UI и UX.
- Как автоматизировать запросы к REST API.
- Как автоматизировать запросы к SOAP API.
- Что такое DDT.

Термины, используемые в лекции

API (Application Programming Interface) — программный интерфейс приложения, с помощью которого одна программа может взаимодействовать с другой. API

позволяет слать информацию напрямую из одной программы в другую, минуя интерфейс взаимодействия с пользователем.

UI-тестирование — этап тестирования ПО, проверяющий графический интерфейс.

Data Driven Testing (DDT) — подход к созданию/архитектуре автоматизированных тестов (юнит, интеграционных, чаще применимо к backend-тестированию), при котором тест умеет принимать набор входных параметров и эталонный результат или эталонное состояние, с которым он должен сравнить результат, полученный в ходе прогонки входных параметров.

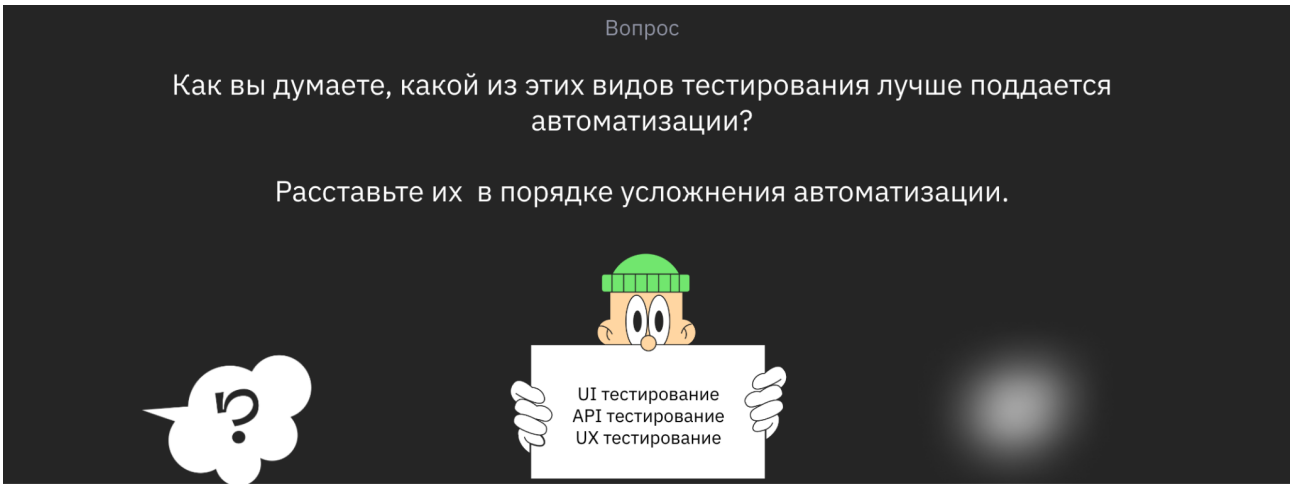
Тестирование API, UI, UX

На этом курсе мы поговорим про тестирование веб-приложений.

Веб-приложения прочно вошли в нашу жизнь. Сейчас они распространены даже больше, чем десктопные.

Как правило, при тестировании веб-приложений проводят:

- **UI-тестирование** — тестирование графического интерфейса. Его задача — проверить функции приложения, имитируя действия пользователей. В процессе тестирования элементы интерфейса проверяют на корректность, вводя данные в приложение через устройства ввода или средства автоматизированного UI-тестирования.
- **UX-тестирование** (юзабилити-тестирование) — тестирование качества интерфейса, его удобства для пользователя. Задача — понять, насколько хорошо, понятно, логично, удобно, правильно разработан ресурс, работают ли на нем все элементы и функции.
- **Тестирование API**. API — это интерфейс, позволяющий двум независимым компонентам программного обеспечения обмениваться информацией. Не у всех веб-приложений есть доступный для тестирования API, но там, где он есть, его обязательно нужно протестировать.



Ответ: удобнее всего автоматизировать тестирование API. Оно сводится к отправке, получению и анализу запросов. Для этого в Python есть специальные библиотеки.

Тестирование UI тоже хорошо поддается автоматизации с использованием библиотеки Selenium. Она позволяет работать с элементами интерфейса — получать их свойства и взаимодействовать с ними.

Хуже всего автоматизируется UX-тестирование, так как оценивать удобство для пользователя автоматически довольно сложно. Лучше, чтобы это делал человек.

На этом уроке мы рассмотрим автоматизацию тестирования UI и API. Начнем с тестирования API.

Виды API (REST, SOAP)

API выполняет роль посредника между внутренними и внешними программными функциями, обеспечивая эффективный обмен информацией. Конечные пользователи могут не замечать работу API, но этот механизм очень важен.

На курсе мы рассмотрим два вида API — REST API и SOAP API.

REST API (от англ. Representational State Transfer, передача состояния представления или передача репрезентативного состояния) — это архитектурный подход, который устанавливает ограничения для API: как они должны быть устроены и какие функции должны поддерживать. Позволяет стандартизировать работу программных интерфейсов, сделать их более удобными и производительными.

В отличие от SOAP API, REST API — не протокол, а список рекомендаций, которым можно следовать или не следовать. Поэтому у него нет собственных методов.

SOAP (от англ. Simple Object Access Protocol, простой протокол доступа к объектам) — это протокол, по которому веб-сервисы взаимодействуют друг с другом или с клиентами.

SOAP API — веб-сервис, использующий протокол SOAP для обмена сообщениями между серверами и клиентами. При этом сообщения должны быть написаны на языке XML в соответствии со строгими стандартами, иначе сервер вернет ошибку.

Ниже образец xml-документа для запроса на формирование электронной подписи. Разделы, отмеченные как !--Optional, могут отсутствовать.

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:sgv="http://www.roskazna.ru/eb/sign/types/sgv">
   <soapenv:Header/>
   <soapenv:Body>
      <sgv:SigningRequestType>
         <sgv:data>DATA</sgv:data>
         <!--Optional:-->
         <sgv:signatureType>cades-bes</sgv:signatureType>
         <!--Optional:-->
         <sgv:detached>false</sgv:detached>
         <!--Optional:-->
         <sgv:xmlPartID>?</sgv:xmlPartID>
         <!--Optional:-->
         <sgv:actor>?</sgv:actor>
         <!--Optional:-->
         <sgv:algorithmId>?</sgv:algorithmId>
         <!--Optional:-->
         <sgv:transforms>cid:470893435143</sgv:transforms>
         <!--Optional:-->
         <sgv:businessProcessId>?</sgv:businessProcessId>
      </sgv:SigningRequestType>
   </soapenv:Body>
</soapenv:Envelope>
```

Сравним SOAP и REST API в таблице:

SOAP API	REST API
Использование XML, WSDL	Ресурс-ориентированная технология
Работа с методами	HTTP-запросы

Поддержка транзакций и уровней безопасности	Для несложной бизнес-модели
Сложнее разрабатывать	Легче разрабатывать

Как мы будем тестировать API в Python?

Нам понадобится библиотека, с помощью которой мы будем отсылать запросы нужного формата к API, получать код ответа и его содержимое. Чтобы проверить результат, зачастую нам нужно будет получить состояние системы, а для этого мы отправим еще один запрос. Так мы проверим, что команда, отправленная первым запросом, корректно выполнена.

Для работы с REST и SOAP API мы можем использовать разные библиотеки:

- для тестирования REST API — **Requests**;
- для тестирования SOAP API — **zeep**.

Библиотека Requests

Установим библиотеку Requests:

```
pip3 install requests
```

Пример запроса:

```
response = requests.get('https://api.github.com')
```

Код ответа можно получить командой:

```
response.status_code
```

Если использовать Response в условных конструкциях, то при получении кода состояния в промежутке от 200 до 400 выведется значение True. В противном случае — False.

Другие запросы:

- requests.post('https://httpbin.org/post', data={'key':'value'})
- requests.put('https://httpbin.org/put', data={'key':'value'})
- requests.delete('https://httpbin.org/delete')

- requests.head('https://httpbin.org/get')
- requests.patch('https://httpbin.org/patch', data={'key':'value'})
- requests.options('https://httpbin.org/get')

Чтобы получить содержимое запроса в байтах, нужно использовать **.content**.

```
1 response = requests.get('https://api.github.com')
2 response.content
```

Использование .content обеспечивает доступ к чистым байтам ответа, то есть к любым данным в теле запроса. Однако зачастую требуется конвертировать полученную информацию в строку в кодировке UTF-8. Response делает это с помощью **.text**.

```
response.text
```

Декодирование байтов в строку требует наличия определенной модели кодировки. Requests попытается узнать текущую кодировку, ориентируясь по заголовкам HTTP. Принудительно указать кодировку можно, добавив **.encoding** перед .text.

```
1 response.encoding = 'utf-8'
2 response.text
```

Часто бывает удобно работать с ответом в формате json. Для этого используется команда:

```
response.json()
```

Тип полученного значения из .json() является словарем. Это значит, что доступ к его содержимому можно получить по ключу.

HTTP-заголовки ответов могут дать нам полезную информацию: тип содержимого ответного пейлоада, а также ограничение по времени для кэширования ответа. Для просмотра HTTP-заголовков используется атрибут:

```
response.headers
```

Наиболее простой способ настроить запрос GET — передача значений через параметры строки запроса в URL. При использовании метода get() данные

передаются в params. Например, чтобы посмотреть на библиотеку Requests, можно использовать Search API на GitHub.

```
response = requests.get( 'https://api.github.com/search
/repositories',
    params={'q': 'requests+language:python'})
```

В некоторых случаях может потребоваться настройка HTTP-заголовка запроса (headers).

Для изменения HTTP-заголовка нужно передать словарь этого HTTP-заголовка в get() с помощью параметра **headers**.

```
import requests

response = requests.get(
    'https://api.github.com/search/repositories',
    params={'q': 'requests+language:python'},
    headers={'Accept': 'application/vnd.github.v3.text-match+json'},
)
```

Запросы **POST**, **PUT** и **PATCH** передают информацию через тело сообщения, а не через параметры строки запроса. Используя **requests**, можно передать данные в параметр **data**.

В свою очередь, **data** использует словарь, список кортежей, байтов или объект файла. Это особенно важно, так как может возникнуть необходимость адаптировать данные, отправляемые с запросом, в соответствии с определенными параметрами сервера.

Например, если тип содержимого запроса application/x-www-form-urlencoded, можно отправить данные формы в виде словаря.

```
requests.post('https://httpbin.org/post', data={'key':'value'})
```

Важно помнить, что многие API требуют аутентификации.

Как правило, вы предоставляете свои учетные данные на сервер, передавая данные через заголовок Authorization или специальный токен авторизации, который выдается при регистрации в API.

Пример API, который требует аутентификации, — Authenticated User API на GitHub. Это конечная точка веб-сервиса, которая предоставляет информацию о профиле аутентифицированного пользователя. Чтобы отправить запрос API-интерфейсу аутентифицированного пользователя, вы можете передать свое имя пользователя и пароль на GitHub через кортеж в get().

```
from getpass import getpass
requests.get('https://api.github.com/user', auth=('username',
getpass()))
```

Пример авторизации с использованием токена:

```
import requests
headers = {'X-Yandex-API-Key': 'b1fb281f-
b5ff-4257-8b94-35f249384a32'}
data = requests.get('https://api.weather.yandex.ru
/v1/forecast?lat=55.466&lon=36.93&extra=true', headers=headers)
print (data.text)
```

Важно отметить, что подготовка к тестированию API обязательно включает в себя изучение документации того API, который мы будем тестировать. Как правило, в ней указаны все нужные параметры и структура запросов.

Пример работы с REST-API

Перейдем на API:Geosearch - MediaWiki и изучим документацию.

Мы хотим сделать запрос ближайших достопримечательностей по координатам и проверить, что ответ содержит некоторую достопримечательность.

💡 В видеолекции присутствует скринкаст, рекомендуемый к просмотру для лучшего понимания.

```python
import requests

S = requests.Session()

def get_sites(lat, long, radius, limit=100):
    URL = "https://en.wikipedia.org/w/api.php"
    params = {
    "action": "query",
    "format": "json",
    "list": "geosearch",
    "gscoord": f"{lat}|{long}",
    "gsradius": f"{radius}",
    "gslimit": f"{limit}"
    }
    r = S.get(url=URL, params=params)
    pages = r.json()["query"]["geosearch"]
    sites = [i["title"] for i in pages]
    return sites

def test_step1():
    assert "One Montgomery Tower" in get_sites("37.7891838",
  "-122.4033522", 100)
```

Мы реализовали тест, который проверяет геопоиск Wiki, а именно проверяет, что в списке достопримечательностей, расположенных вблизи координат, заданных в примере, присутствует достопримечательность **One Montgomery Tower**.

Мы можем составить список достопримечательностей, которые обязательно должны быть отражены в Википедии для более полной проверки.

Библиотека zeep для SOAP

Zeep проверяет документ WSDL и генерирует соответствующий код для использования служб и типов в программе. Это обеспечивает простой в использовании программный интерфейс запросов к серверу SOAP.

Устанавливается командой:

```
pip3 install zeep
```

Можно выполнить установку из Pycharm.

Простой пример работы с zeep:

```python
from zeep import Client

client = Client('http://www.webservicex.net/ConvertSpeed.asmx?WSDL')
result = client.service.ConvertSpeed(
    100, 'kilometersPerhour', 'milesPerhour')

assert result == 62.137
```

Что происходит в примере?

Для начала работы SOAP нужно знать URL-адрес WSDL.

Рассмотрим еще один простой пример сервиса со следующим адресом wsdl: http://www.soapclient.com/xml/soapresponder.wsdl. Описание структуры можно получить, открыв адрес WSDL в браузере.

```
-<definitions name="SoapResponder" targetNamespace="http://www.SoapClient.com/xml/SoapResponder.wsdl">
  -<types>
     <schema targetNamespace="http://www.SoapClient.com/xml/SoapResponder.xsd"> </schema>
   </types>
  -<message name="Method1">
     <part name="bstrParam1" type="xsd:string"/>
     <part name="bstrParam2" type="xsd:string"/>
   </message>
  -<message name="Method1Response">
     <part name="bstrReturn" type="xsd:string"/>
   </message>
  -<portType name="SoapResponderPortType">
    -<operation name="Method1" parameterOrder="bstrparam1 bstrparam2 return">
       <input message="tns:Method1"/>
       <output message="tns:Method1Response"/>
     </operation>
   </portType>
```

Однако формат xml не очень удобен для чтения человеком.

Чтобы получить удобное описание, нужно выполнить команду

```
python3 -m zeep http://www.soapclient.com/xml/soapresponder.wsdl
```

И мы получим список методов с аргументами:

```
Bindings:
    Soap11Binding: {http://www.SoapClient.com/xml/SoapResponder.wsdl}SoapResponderBinding

Service: SoapResponder
    Port: SoapResponderPortType (Soap11Binding: {http://www.SoapClient.com/xml/SoapResponder.wsdl}SoapResponderBinding)
        Operations:
            Method1(bstrParam1: xsd:string, bstrParam2: xsd:string) -> bstrReturn: xsd:string
```

Теперь мы можем инициализировать клиент zeep с помощью URL-адреса WSDL, а затем просто вызывать методы, описанные в схеме.

Дополнительные параметры можно передать с использованием Settings. Например, можно ослабить требования к структуре ответа:

```
1 settings = Settings(strict=False)
2 client = Client(wsdl=wsdl, settings=settings)
```

Рассмотрим более сложный пример. Напишем тест, который выполняет проверку электронной подписи, через тестовый SOAP-сервис КриптоПро, доступный по адресу: http://dss.cryptopro.ru/verify/service.svc?wsdl.

Откроем этот адрес в браузере:

Видим описание сервисов. Получим это описание в более удобном виде командой:

```
python3 -m zeep http://dss.cryptopro.ru/verify/service.svc?wsdl
```

Мы можем найти там описание операции проверки подписи:

```
VerifySignature(signatureType: ns2:SignatureType, document:
xsd:base64Binary,
verifyParams: ns4:ArrayOfKeyValueOfVerifyParamsstring1Iy7z97I) ->
VerifySignatureResult: ns2:VerificationResult
```

Нас будут интересовать только первые два аргумента.

Напишем тест, который будет проверять валидность подписи при помощи SOAP:

```
from zeep import Client
wsdl = "http://dss.cryptopro.ru/verify/service.svc?wsdl"
sign =
  """MIINaQYJKoZIhvcNAQcCoIINWjCCDVYCAQExDjAMBggqhQMHAQECAgUAMBIGCSqGS
.....................................
yTk2nHb/8qkb5z+nZRqOj5Sp/uBvMRzD8="""
client = Client(wsdl=wsdl)

def test_step1():
    assert client.service.VerifySignature('CMS', sign)["Result"]
```

Что такое DDT

Data Driven Testing (DDT) — подход к созданию/архитектуре автоматизированных тестов (юнит, интеграционных, чаще всего применимо к backend-тестированию), при котором тест умеет принимать набор входных параметров и эталонный результат или эталонное состояние, с которым он должен сравнить результат, полученный в ходе прогонки входных параметров (Data Driven Testing).

Такое сравнение и есть assert такого теста. При том как часть входных параметров могут передаваться опции выполнения теста или флаги, которые влияют на его логику.

Особый плюс хорошо спроектированного DDT — возможность ввести входные значения и эталонный результат в виде, удобном для всех ролей на проекте: от мануального тестировщика до менеджера проекта (тест-менеджера) и даже product owner'а (на практике автора такие случаи были).

Соответственно, когда вы способны загрузить мануальных тестировщиков увеличением покрытия и увеличением наборов данных, это удешевляет тестирование. Кроме того, удобный и понятный формат позволяет более наглядно видеть, что покрыто, а что нет. Это, по сути, и есть документация тестирования. К примеру, это может быть XLS-файл с понятной структурой (хотя чаще всего properties файла достаточно).

Или же создатель такого теста может дать коллегам workflow, по которому можно просто подготовить эталонные значения. То есть желательно избежать зависимости в создании наборов данных от программистов и автоматизаторов — любой на проекте должен суметь их подготовить.

Переделаем рассмотренный ранее пример на использование DDT.

conftest.py

```python
import pytest

@pytest.fixture()
def coord1():
    return "37.7891838", "-122.4033522"

@pytest.fixture()
def text1():
    return "One Montgomery Tower"
```

test_1.py

```python
import requests

S = requests.Session()

def get_sites(lat, long, radius, limit=100):
    URL = "https://en.wikipedia.org/w/api.php"
    params = {
    "action": "query",
    "format": "json",
    "list": "geosearch",
    "gscoord": f"{lat}|{long}",
    "gsradius": f"{radius}",
    "gslimit": f"{limit}"
    }
    r = S.get(url=URL, params=params)
    pages = r.json()["query"]["geosearch"]
    sites = [i["title"] for i in pages]
    return sites

def test_step1(text1, coord1):
    assert text1 in get_sites(coord1[0], coord1[1], 100)
```

Мы получили тест, изменить который можно, не меняя код. Достаточно поменять данные, которые возвращают фикстуры в conftest.py

Итоги

На этом уроке мы узнали:

- Особенности тестирования API, UI и UX.

- Как автоматизировать запросы к REST API.
- Как автоматизировать запросы к SOAP API.
- Что такое DDT.

Что можно почитать еще?

1. Различия REST и SOAP
2. HTTP-библиотека requests в Python — описание библиотеки Requests
3. Тестирование на основе данных

Используемая литература

1. Библиотека Requests: HTTP for Humans
2. Zeep: Python SOAP client — документация модуля zeep
3. Data-Driven Testing
4. Data Driven Testing (7 примеров использования с рабочих проектов)