# Инфо

QA – quality assurance

Разные подходы к QA:

1) agile

2) devops –

## Виды тестирования

<https://www.youtube.com/watch?v=TLccnKIMggA>

Статическое тестирование static testing

Проверка кода на наличие синтаксических ошибок или несоответсвие принятым нормам.

Тестируется с помощью eslint или встроенными как в typescript.

Unit Testing

Проверить корректность работы отдельных функций, методов или классов.

def test\_add\_numbers():

assert add(2, 3) == 5 # Проверяем функцию сложения

Интеграционное тестирование (Integration Testing)

Проверить взаимодействие между модулями, сервисами или системами.

Часто требуют настройки тестового окружения (например, Docker).

End-to-end тестирование (e2e)

Полноценное тестирование приложения

Contract test / API test / API-integration test

Тестирование сервисов по url эндпоинтам или по сваггеру на соответствие ожидаемого поведения.

Функциональное тестирование (Functional Testing / UI Testing)

Проверить, соответствует ли система требованиям с точки зрения пользователя.

Тестируется графический интерфейс (кнопки, формы, навигация).

Нагрузочное тестирование (Performance Testing)

Проверить, как система ведет себя под нагрузкой.

Тестирование безопасности (Security Testing)

Найти уязвимости и защитить систему от атак.

Тестирование баз данных (Database Testing)

Убедиться, что данные сохраняются, обновляются и удаляются корректно.

Тестирование в CI/CD

Автоматизировать запуск тестов при каждом изменении кода.

## Концепции

Mocking – когда тебе нужно протестирует функцию, которая что то делает с к примеру БД, но ты не хочешь подключать настоящую БД из за добавления сложности. Вместо этого можно симулировать БД подменой – эта подмена и есть mock (fake).

## cURL

<https://curl.se/>

documentation -> curl tool documentation

Что такое URL

<https://www.ietf.org/rfc/rfc3986.txt>

## OpenAPI (Swagger)

Редактор сваггер документаций

<https://www.bairesdev.com/tools/json2yaml/>

## JSON Schema

Json schema – это метаданные json файла

Код пример:

pip install jsonschema

import jsonschema

from jsonschema import validate

# Определяем схему

schema = {

"type": "object",

"properties": {

"name": {"type": "string"},

"age": {"type": "integer", "minimum": 18},

},

"required": ["name", "age"]

}

# Пример данных

data = {

"name": "Nikolay Sobolev",

"age": 19

}

# Валидация данных

try:

validate(instance=data, schema=schema)

print("Данные валидны")

except jsonschema.exceptions.ValidationError as err:

print(f"Ошибка валидации: {err.message}")

# Тестирование

Легко реализовать с помощью pytests и requests в python-е

# Property-based автоматическое тестирование

Проперти бейсд тестирование происходит благодаря извлечению схемы API сервиса (swagger)

Схема API – Это некоторый файл json или yaml, который описывает структуру проекта по его эндпоинтам и генерируется автоматически и изменяется автоматически по мере изменения кода. Также по мимо эндпоинтов в нем содержаться типы данных для запроса к каждому эндпоинту и то как выглядят ответы на запрос. Они бывают в yaml или json форматах.

Пример

**swagger.yaml**

openapi: 3.0.1

info:

title: Настройка промышленного протокола Modbus

version: '1.0'

paths:

"/modbus/connections":

get:

tags:

- 1. Соединения

summary: Получить список соединений

description: "<b>plantAlias</b> - псевдоним промышленного объекта"

operationId: GetAllConnections

parameters:

- name: plantAlias

in: header

required: true

schema:

type: string

responses:

'200':

description: OK

content:

application/json:

………….

NOTE:

Такие схемы создаются с помощью Swagger (OpenAPI), НО они подходят только для REST архитектур. Swagger не может создать такие схемы для gRPC и GraphQL, а также для AsyncAPI.

# \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Стэк

## Тестирование

**Статическое тестирование**

### eslint

Код

В существующем js/ts проекте:

1) Установка зависимостей eslint

npm install --save-dev eslint @typescript-eslint/parser @typescript-eslint/eslint-plugin

2) Создать конфиг для eslint

При этом правила проверки и нормы также записываются в конфиге

eslint.config.mjs

import { defineConfig } from "eslint/config";

import typescriptEslint from "@typescript-eslint/eslint-plugin";

import globals from "globals";

import tsParser from "@typescript-eslint/parser";

import path from "node:path";

import { fileURLToPath } from "node:url";

import js from "@eslint/js";

import { FlatCompat } from "@eslint/eslintrc";

const \_\_filename = fileURLToPath(import.meta.url);

const \_\_dirname = path.dirname(\_\_filename);

const compat = new FlatCompat({

baseDirectory: \_\_dirname,

recommendedConfig: js.configs.recommended,

allConfig: js.configs.all

});

export default defineConfig([{

extends: compat.extends("eslint:recommended", "plugin:@typescript-eslint/recommended"),

plugins: {

"@typescript-eslint": typescriptEslint,

},

languageOptions: {

globals: {

...globals.browser,

},

parser: tsParser,

ecmaVersion: "latest",

sourceType: "module",

},

rules: {

"no-unused-vars": "off",

"no-var": "error",

"@typescript-eslint/no-unused-vars": ["off"],

"no-delete-var": "off",

},

}]);

ЕСЛИ ПРОБЛЕМЫ С МИГРАЦИЕЙ другого формата конфига

npx @eslint/migrate-config .eslintrc.json

npm install globals @eslint/js @eslint/eslintrc –D

npm audit fix

3) Проверка файлов

Всех

npx eslint .

конкретные

npx eslint code-analysis.ts multiply.js multiply.ts some-function.js

**Юнит тесты**

### jasmine

<https://jasmine.github.io/>

Jasmine используется для проверки выходных значений проверяемой функции. Соответствуют ли они должному поведению или нет. Базовый функционал представлен ниже, здесь представлена проверка того, что функция правильно складывает числа положительные и отрицательные, на основе ожидаемого поведения (3 и 7).

describe("sum()", () => {

it("should sum two positive numbers", () => {

const actual = sum(1, 2);

const expected = 3;

expect(actual).toEqual(expected);

});

it("should sum positive and negative numbers", () => {

const actual = sum(-3, 10);

const expected = 7;

expect(actual).toEqual(expected);

});

});

код

1) установка зависимостей

npm install --save-dev jasmine-spec-reporter

npm install --save-dev jasmine

2) Создание папок и файлов для jasmine

jasmine init

### jest

<https://jestjs.io/docs/getting-started>

**E2E**

### Cypress

<https://www.cypress.io/>

**API-test**

### Dredd

Берёт сваггер, извлекает из него всю информацию и делает автоматические тесты. Захардкожен.

### Schemathesis

<https://habr.com/ru/articles/478114/>

<https://schemathesis.readthedocs.io/en/stable/>

Основан на Hypothesis

Делает умные и property based тесты основываясь на openAPI схеме проекта.

Может работать автоматически из командной строки и в таком варианте похожа на dredd.

Но вообще это библиотека на пайтон, которая может взаимодействовать с pytest и писать кастомные тесты и встраиваться в cicd пайплайны.

Как работает Schemathesis:

1. Получает json или yaml схему из сваггера
2. На её основе создаёт json schema, которая описывает какие типы могут быть в полях
3. С помощью hypothesis создаёт множество property-based тестов
4. Используем эти тестах в Ci тестах

Установка:

python –m install schemathesis

### Hypothesis

Представляет генератор различных объектов, основываясь на типе данных объектов. Он помогает тестировать обычные функции генерируя тысячи различных вохможных массивов, строк итд чтобы подставить их в функцию и проверить, что функция работает для многих вариантов строк.

Обычно используется вместе с pytests для тестирования функций.

Пример:

import json

from hypothesis import given, strategies as st

@given(st.text())

def test\_json\_roundtrip(s):

asserts = json.loads(json.dumps(s))

Можно показать больше информации так в консоли.

pytests –hypothesis-show-statistics

### Pytest

#### Info

Описание:

Ультимативный инструмент, который может делать любые тесты. Но надо писать много кода, и он автоматически не генерирует тесты.

Как работает библиотека:

Pytest автоматически сканирует директорию командной строки и ищет все файлы, название которых начинается с test\*.py и внутри этих файлов ищет функции которые начинаются с test\* и запускает их

Запуск и флаги:

pytest -v –s - запуск

-v – показывают какие функции прошли проверку

-s – можно видеть свои принты

Фикстуры:

#### работа

Пример программы которая проверяет api через сваггер.

import requests

ENDPOINT = 'https://todo.pixegami.io'

def test\_success\_call():

r = requests.request('GET', ENDPOINT, verify=False)

assert r.status\_code == 200

pass

def test\_create\_task():

payload = {

"content": "WUSSUUUUUP",

"user\_id": "Crazy8",

"is\_done": False

}

r = requests.request('PUT', ENDPOINT+'/create-task', json = payload, verify=False)

data = r.json()

assert r.status\_code == 200

print(data)

task\_id = data['task']['task\_id']

r2 = requests.request('GET', ENDPOINT+f'/get-task/{task\_id}', verify=False)

data2 = r2.json()

assert r2.status\_code == 200

assert data2['content'] == payload['content']

print(data2)

### Postman-Newman

БЕЗ postman коллекции не работает.

Позволяет создать в postman коллекцию тестов и встроить автоматическую проверку этих тестов через CLI Newman ( запуск коллекции postman из консоли ) И ВНЕДРИТЬ это в github actions чтобы создать CI/CD пайпайн проверки ошибок если разработчик делает push например.

### Swagger test templates + Mocha

Swagger test templates берет сваггер, извлекает всю информацию и эндпоинты. На основе этого автоматически создает множетсво .js файлов – тестов каждого эндпоинта, готовых к запуску.

Запустить файлы с тестами можно с помощью Mocha.

## Package managers

### UV

<https://docs.astral.sh/uv/>

Тоже самое что pip но быстрее и может schemathesis запускать без установки самого schemathesis.

Установка на виндовс

powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"

Запуск schemathesis

uvx schemathesis run <https://example.schemathesis.io/openapi.json>

# Pytest

## fixture

**Опредление**

(своими словами) Фикстура - функция / БД , которая может вызываться во время теста целевой функции и может быть доступна для других тестов в зависимости от настроек. То есть это внешние данные или внешний фрагмент кода, который может вызываться для тестирования в целевой функции как ресурс, тк для теста часто нужны некоторые сторонние данные (контекст).

(документация + вики) Test fixture (также называемая «тестовым контекстом») используется для настройки состояния системы и входных данных, необходимых для выполнения теста. Фикстура обеспечивает определенный, надежный и последовательный контекст для тестов. Это может включать среду (например, базу данных, настроенную с известными параметрами) или контент (например, набор данных).

@pytest.fixture

def add():

return 1

def test\_3(add):

assert 1 + add == 2

**Зависимости фикстур**

Могут быть сложные зависимости между фикстурами:

@pytest.fixture

def order():

return []

@pytest.fixture

def append\_first(order):

order.append(1)

@pytest.fixture

def append\_second(order, append\_first):

order.extend([2])

@pytest.fixture(autouse=True)

def append\_third(order, append\_second):

order += [3]

def test\_order(order):

assert order == [1, 2, 3]

При вызове ордер создается пустой массив в test\_order, и после того как основные фикстуры все сделали return вызываются фикстуры с autouse = True:

1. вызывается append\_third
2. сначала импортируется ордер
3. потом импортируется append\_second
4. для того чтобы импортировать append\_second нужно чтобы append\_second выполнила свой алгоритм
5. в append\_second также импортируется ордер, потом импортируется append\_first
6. итд пока не зависимости не достигнут конца
7. К этому моменту в test\_order ордер будет поочереди заполнен числами с помощью фикстур, которые ждали друг друга.

**Область действия фикстур**

scope

@pytest.fixture(scope=)

def add():

return []

def test\_3(add):

add.append(1)

def test\_4(add):

add.append(1)

assert len(add) == 1

## paramatarize

Встроенный декоратор pytest.mark.parametrize позволяет параметризовать аргументы для тестовой функции. Параметризация происходит вручную (неэффективно).

import pytest

@pytest.mark.parametrize("test\_input,expected", [("3+5", 8), ("2+4", 6), ("6\*9", 42)])

def test\_eval(test\_input, expected):

assert eval(test\_input) == expected

# Hypothesis

## Определение

Гипотезис базируется на идее того, что

1) мы передаём свою функцию обёртки гипотезиса

2) итеративно обёртка тестирует функцию, генерируя и передавая ей в аргумент объекты

3) так варьируя передаваемый аргумент мы может наткнуться на потенциальный баг

Так например мы можем протестировать нашу функцию для набора целых чисел:

@given(..smth…)

def test\_integers(n):

print(f"called with {n}")

assert isinstance(n, int)

test\_integers()

Здесь @given — это стандартная точка входа в Hypothesis. Она принимает стратегию, которая описывает тип входных данных, которые должна принимать декорированная функция. Когда мы вызываем test\_integers, Hypothesis генерирует набор целых чисел.

Инфо

По умолчанию Hypothesis генерирует 100 случайных примеров.

## Генерация

В зависимости от функции пользователь должен сам определить тип генерируемых объектов (или как в документации Domain):

@given( strategy.domain() )

def func(sample):

@given(strategy.lists(stategy.integers()))

def func(sample):

given - декоратор который создаёт цикл, на каждой итерации которого стратегия генерирует sample и передаёт в функцию.

strategy - стратегия это алгоритм который “знает” как генерировать сэмплы того или иного домена и возвращает эти семплы в ретёрн.

domain - тип объекта ( integer, string, list[integers], list[strings] ).

Вывести примеры генерации:

from hypothesis import strategy as st

print(st.lists(st.integers() | st.floats(allow\_nan=False)).example())

## Настройка генерации

Тк генерирует сэмплы стратегия, то настройка генеарции происходит через параметры стратегии.

Список всех стратегий и аргументов-настроек для них

[Strategies Reference - Hypothesis 6.135.2 documentation](https://hypothesis.readthedocs.io/en/latest/reference/strategies.html)

1) Доступные настройки для генерации целых чисел

@given(st.integers(0, 200))

def test\_integers(n):

assert n < 50

2) Доступные настройки для генерации текста

@given(strategies.text(alphabet=’abc’, min\_size=0, max\_size=None)

def test\_strs(sample):

## Настройка проведения тестов

<https://hypothesis.readthedocs.io/en/latest/reference/api.html#hypothesis.settings.max_examples>

Указать количество семплов которые сгенерируются в конкретном тесте:

from hypothesis import given, settings, strategies as st

@given(st.integers())

@settings(max\_examples=200)

def runs\_200\_times\_instead\_of\_100(n):

pass

Изменить настройки глобально для всех тестов:

from hypothesis import HealthCheck, settings

settings.register\_profile("fast", max\_examples=10)

## Фичи

Добавить мануально несколько сэмплов и протестировать их перед основной генерацией:

@example("Hello world")

@example("some string with special significance")

@given(st.text())

def test\_strings(s):

pass

Добавление своих собственных стратегий

import pytest

from hypothesis import assume, given, strategies as st

@st.composite

def sums\_to\_n(draw, n=1):

lst = draw(st.lists(st.floats(0, 1), min\_size=1))

assume(sum(lst) > 0)

return [f / sum(lst) \* n for f in lst]

@given(sums\_to\_n(10))

def test(lst):

assert sum(lst) == pytest.approx(10)

Генерация данных прямо по середине теста

import pytest

from hypothesis import given, strategies as st

@given(st.data())

def test(data):

lst = data.draw(st.lists(st.floats(0.001, 1), min\_size=1))

lst = [f / sum(lst) for f in lst]

assert sum(lst) == pytest.approx(1)

5) Тестирование на основе целевых свойств

Тестирование на основе целевых свойств сочетает в себе преимущества как тестирования на основе поиска, так и тестирования на основе свойств. Вместо того чтобы быть полностью случайным, целевой PBT использует компонент на основе поиска, чтобы направлять генерацию входных данных к значениям, которые имеют более высокую вероятность фальсификации свойства. Это более эффективно исследует входное пространство и требует меньше тестов для поиска ошибки или достижения высокой уверенности в тестируемой системе, чем случайный PBT. (Лёшер и Сагонас)

Это не всегда хорошая идея — например, вычисление метрики поиска может занять время, которое лучше потратить на запуск более равномерно-случайных тестовых случаев, или ваша целевая метрика может случайно увести гипотезу от ошибок — но если есть естественная метрика, такая как «ошибка с плавающей точкой», «коэффициент загрузки» или «длина очереди», мы рекомендуем вам поэкспериментировать с целевым тестированием.

from hypothesis import given, strategies as st, target

@given(st.floats(0, 1e100), st.floats(0, 1e100), st.floats(0, 1e100))

def test\_associativity\_with\_target(a, b, c):

ab\_c = (a + b) + c

a\_bc = a + (b + c)

difference = abs(ab\_c - a\_bc)

target(difference) # Without this, the test almost always passes

assert difference < 2.0

## Распределение генерации

Hypothesisследует философии, согласно которой недопустимо разглашать распределение генерации чисел, текстов и прочих типов данных. Пользователь может контролировать диапазон и количество тестов, но распределение или же то, какие именно семплы и как часто будут использоваться в тестах, должно быть недоступно ему. Это позволяет генерировать более “неожиданные” кейсы и убирает элемент предвзятости пользователя, чтобы проверить его функции на действительно всех edge кейсах.

Поэтому нельзя заранее знать какие значения будут генерироваться в тестах, но можно задать диапазон и другие параметры.

## Shrinking (минимизация)

Шринкинг - это механизм гипотезис, который активируется при падении теста. Когда гипотезис обнаруживает ошибку, то он проводит дополнительные проверки, изменяя и минимизируя запрос вызвавший ошибку, чтобы найти конкретный элемент, вызывающий ошибку. Например если в текстовом аргументе есть арабская буква, то гипотезис найдет её и скажет что именно она вызывает ошибку.

Или пример функции вызывающую ошибку при обнаружении отрицательного числа во входном списке:

from hypothesis import given, strategies as st

@given(st.lists(st.integers()))

def test\_no\_negative\_numbers(xs):

for x in xs:

assert x >= 0

Гипотезис находит минимальное но вызывающее ошибку значение

Ответ пайтеста:

FAILED hypo.py::test\_no\_negative\_numbers - assert -1 >= 0

=================== 1 failed, 2 passed, 1 warning in 1.14s =

## Советы

Следует проводить Round Tip тестирование, которое проверяет корректность сериализацию и десериализацию:

decode(encode(x)) == x

json.loads(json.dumps(data)) == data

Пример теста:

import base64

from hypothesis import given, strategies as st

@given(st.binary()) # генерируем произвольные байты

def test\_base64\_round\_trip(data: bytes):

encoded = base64.b64encode(data)

decoded = base64.b64decode(encoded)

assert decoded == data

# Schemathesis

Документация

[Welcome to Schemathesis! — Schemathesis 3.39.16 documentation](https://schemathesis.readthedocs.io/en/stable/index.html)

Осуществляет множество запросов по адресу API, если у него есть swagger и openAPI схема. На основе этой схемы он формирует множество вариантов тестов.

Данные, их диапазон и их распределение, которые генерируются в процессе тестов реализуются полностью благодаря hypothesis. Для уточнения информации о данных, их распределении смотреть документацию hypothesis.

## Какие данные генерируются

Schemathesis использует hypothesis стратегии для генерации данных, при этом он использует все значения по умолчанию, включая диапазоны генерации. Изменить из напрямую нельзя, поэтому диапазоны генерации всегда одинаковые.

Таблица диапазонов согласно документации hypothesis

|  |  |  |
| --- | --- | --- |
| Тип данных | Стратегия | Диапазон по умолчанию |
| None | none() | Только None. |
| Булевы значения | booleans() | True или False. |
| Целые числа | integers() | Весь диапазон int в Python (зависит от платформы, обычно от -2\*\*63 до 2\*\*63-1). |
| Вещественные числа | floats() | Почти все float (включая ±inf, NaN, субнормальные), кроме исключений в IEEE 754. |
| Комплексные числа | complex\_numbers() | Любые комплексные числа с min\_magnitude=0 (без ограничений на модуль). |
| Строки | text() | Unicode-символы (кроме суррогатных), длина от 0 до None (но разумно ограничена). |
| Бинарные данные | binary() | Байты длины от 0 до None (разумно ограничено). |
| Списки | lists(elements) | Длина от 0 до None (разумно ограничено). |
| Словари | dictionaries(keys, values) | Длина от 0 до None (разумно ограничено). |
| Даты | dates() | От datetime.date.min (1-01-01) до datetime.date.max (9999-12-31). |
| Время | times() | От datetime.time.min (00:00:00) до datetime.time.max (23:59:59.999999). |
| Дата-время | datetimes() | От datetime.datetime.min (1-01-01 00:00:00) до datetime.datetime.max (9999-12-31 23:59:59.999999). |
| UUID | uuids() | Все версии UUID (если не указана version). |
| Email | emails() | RFC-совместимые адреса (например, user@example.com). |

*Часто встречаемые типы данных в openapi схемах*

Базовые типы (по спецификации JSON Schema)

string – строка (может включать форматы email, date, uuid и др.).

number – число с плавающей запятой (включает float и double).

integer – целое число (может быть int32, int64).

boolean – логическое значение (true/false).

null – явное указание на null.

Расширенные типы (форматы OpenAPI)

date – строка в формате YYYY-MM-DD (например, "2025-06-09").

date-time – строка в формате ISO 8601 ("2025-06-09T14:00:00Z").

email – валидный email ("user@example.com").

uuid – UUID-строка ("550e8400-e29b-41d4-a716-446655440000").

uri / url – ссылка ("https://example.com").

binary – бинарные данные (например, для загрузки файлов).

byte – base64-кодированные данные ("U3dhZ2dlciByb2Nrcw==").

Составные типы

object – словарь/хэш (например, {"name": "John", "age": 30}).

array – массив значений (например, [1, 2, 3]).

## Как работает генерация данных

**Как парсит openAPI схему**

<https://schemathesis.readthedocs.io/en/stable/data-generation.html>

Schemathesis преобразует схемы Open API в совместимые схемы JSON и передает их в hypothesis-jsonschema, которая генерирует данные для этих схем.

в hypothesis-jsonschema

<https://pypi.org/project/hypothesis-jsonschema/>

**Алгоритм создания сэмплов теста**

Для каждого эндпоинта API Schemathesis следует строгому порядку фаз при генерации семплов.

Вот все фазы по порядку выполнения:

1. Явные примеры (explicit). Сначала алгоритм использует предопределенные примеры из схемы и проводит тесты.
2. Повторный запуск известных сбоев (reuse): Потом использует ранее неудавшиеся тестовые случаи, чтобы проверить исправилась ошибка или нет.
3. Случайная генерация (generate): После этих проверочных фаз наступает основная – алгоритм генерирует набор семплов на основе схемы.
4. Минимизация (shrink): Если попалась ошибка, то алгоритм генерирует минимизированные версии того же самого ошибочного семпла, чтобы определить в чем именно ошибка

## Валидация ответа от сервера

## CLI

Схематезис имеет интерфейс в консоли и позволяет так через консоль тестировать сайты для которых написаны документации swagger 3.0.х.:

schemathesis run <http://localhost:8080/openapi.json>

А также swagger 3.1.х. в экспериментальном режиме:

schemathesis run http://localhost:8080/openapi.json --experimental=openapi-3.1

### флаги

А также существует много флагов контролирующих количество проводимых тестов итд

--hypothesis-max-examples=1000

## Python

### 101

Более гибкое использование схематезис представлено в виде пайтон библиотеки.

Точно такой же функционал как и при schemathesis run <http://localhost:8080/openapi.json> можно реализовать как:

import schemathesis

schema = schemathesis.from\_uri('http://localhost:8080/openapi.json',)

@schema.parametrize()

def test\_app(case):

    case.call\_and\_validate()

|  |
| --- |
| NOTE:  parametaize() в pytest и в schemathesis это РАЗНОЕ |

Для схем 3.1. достаточно просто добавить в начале:

schemathesis.experimental.OPEN\_API\_3\_1.enable()

### Импорт схемы

Стандартный импорт из работающего онлайн или локально веб приложения имеющего сваггер:

from hypothesis import settings

import schemathesis

schemathesis.experimental.OPEN\_API\_3\_1.enable()

schema = schemathesis.from\_uri('http://localhost:8000/openapi.json',)

Удобный импорт схемы прямо из объекта фастапи и фласк:

from project import app

# WSGI

schema = schemathesis.from\_wsgi("/api/openapi.json", app)

# Or ASGI

schema = schemathesis.from\_asgi("/api/openapi.json", app)

### Параметры запросов в сеть

Отключение проверки сертификата TLS

Иногда во время тестирования требуется отключить проверку TLS тестируемой службы.

import schemathesis

schema = schemathesis.from\_uri("http://127.0.0.1/schema.json")

@schema.parametrize()

def test\_api(case):

# If you need `response`

response = case.call(verify=False)

# Alternatively if you don't need `response`

case.call\_and\_validate(verify=False)

### Выбор эндпоинтов

C помощью include exclude name можно контролировать то какие эндпоинты мы проверяем.

@schema.exclude(name = 'POST /todos2/').parametrize()

def test\_app(case):

case.call\_and\_validate()

### Генерация

После извлечения схемы openapi сгенерировать данные можно следующим образом:

schema = schemathesis.from\_uri('http://localhost:8080/openapi.json',)

@schema.parametrize()

def test\_app(case):

    case.call\_and\_validate()

Вывести примеры генерации:

operation = schema["/pet"]["POST"]

strategy = operation.as\_strategy()

print(strategy.example())

### Настройки генерации

* В schemathesis для генерации используется strategy из hypothesis.
* По-умолчанию, при парсинге схемы и получении типов параметров каждого эндпоинта, используется стандартные настройки стратегий hypothesis для проведения тестов.

**Brief**

Напрямую управлять диапазоном генерации стратегий в schemathesis нельзя, несмотря на то, что он использует hypothesis стратегии (которые можно настраивать). Для этого нет интерфйса, если несчитать фильтры и друие костыли.

Но задать диапазон возможно через задание в openapi схеме min и max значений. Пример такой схемы:

{

"openapi": "3.0.0",

"info": {

"title": "User Profile API",

"version": "1.0.0"

},

"paths": {

"/users": {

"post": {

"requestBody": {

"content": {

"application/json": {

"schema": {

"type": "object",

"properties": {

"age": {

"type": "integer",

"minimum": 18,

"maximum": 120,

"example": 25

},

"username": {

"type": "string",

"minLength": 3,

"maxLength": 20,

"pattern": "^[a-zA-Z0-9\_]+$",

"description": "Только латинские буквы, цифры и \_",

"example": "john\_doe"

},

"email": {

"type": "string",

"format": "email",

"maxLength": 100,

"example": "user@example.com"

},

"password": {

"type": "string",

"minLength": 8,

"maxLength": 32,

"pattern": "^(?=.\*[A-Z])(?=.\*\\d).+$",

"description": "Пароль должен содержать минимум 1 заглавную букву и 1 цифру",

"example": "Password123"

},

"is\_active": {

"type": "boolean",

"default": true

},

"score": {

"type": "number",

"minimum": 0.0,

"maximum": 10.0,

"example": 7.5

},

"tags": {

"type": "array",

"items": {

"type": "string",

"maxLength": 15

},

"maxItems": 5,

"example": ["admin", "user"]

}

},

"required": ["username", "email", "password"]

}

}

}

}

}

}

}

}

Разве, что там есть schema.as\_strategy(), что то такое, но тоже это не серьезно.  
  
НО в целом возможно это не такая большая проблема, тк позволяет вашему сервису быть устойчивым к любому вводу.

#### Негативное / позитивное тестирование

Негативное тестирование меняет некоторые типы на неправильные, убирает обязательные значения, добавляет случайные строки

А ТАКЖЕ СЧИТАЕТ ПОГРАНИЧНЫЕ ЗНАЧЕНИЯ КАК НЕГАТИВНЫЕ например тип параметра str (без каких либо ограничений) то пустая строка «» будет считаться негативным значением параметра str какого то фига.

**generation.mode**

Type: String  
Default: "all"

Test data generation mode. Controls whether Schemathesis generates valid data, invalid data, or both.

Possible values:

* "positive": Generate only valid data according to the schema
* "negative": Generate only invalid data to test error handling
* "all": Generate both valid and invalid data

[generation]

mode = "negative"

max-examples = 12

Позитивного тестирования лучше (НЕ ВСЕГДА УЧИТЫВАЕТ ДИАПАЗОНЫ форматов):

[generation]

mode = "positive"

max-examples = 200

[phases.examples]

enabled = false

[phases.coverage]

enabled = false

[phases.fuzzing]

enabled = true

[phases.stateful]

enabled = false

Более рандомное позитивное тестирование (лучше его брать для позитивного тестирования):

[generation]

mode = "positive"

max-examples = 200

[phases.examples]

enabled = false

[phases.coverage]

enabled = true

[phases.fuzzing]

enabled = true

[phases.stateful]

enabled = false

Негативное тестироавние:

[generation]

mode = "negative"

max-examples = 200

[phases.examples]

enabled = false

[phases.coverage]

enabled = true

[phases.fuzzing]

enabled = true

[phases.stateful]

enabled = false

#### Фазы генерации

1. Фаза examples в первую очередь проверяет примеры напрямую записанные в схеме (пропущенные значения заполняются случайно)

[phases.examples]

enabled = false

1. Фаза coverage добавляет вариативности, добавляет варианты с разными методами, генерирует данные выходящие за пределы указанных диапозонов.

[phases.coverage]

enabled = false

1. Генерирует значения в соответствии со схемой. В случае негативного тестирования пытается неправильные генерировать.

[phases.fuzzing]

enabled = true

1. Чото сложное

[phases.stateful]

enabled = false

#### Настройка генерации строк

Настройка происхоит в schemathesis.toml

**generation.allow-x00**

Type: Boolean  
Default: true

Controls whether to allow the generation of 'NULL' bytes (0x00) within strings. Some systems may not handle these bytes correctly.

[generation]

allow-x00 = false

**generation.exclude-header-characters**

Type: String  
Default: \r\n

All characters from the given strings will be excluded from generated header values.

[generation]

exclude-header-characters = "\r\nABC"

**generation.codec**

Type: String  
Default: null

The codec used for generating strings. Defines the character encoding for string generation.

[generation]

codec = "ascii"

#### Добавление своих стратегий

<https://schemathesis.readthedocs.io/en/stable/python.html#tests-configuration>

Можно добавлять свои стратегии в тест, НО через спеиальную обёрнутую @schema.given(st. …)

from hypothesis import strategies as st

schema = ... # Load the API schema here

@schema.include(path="/api/auth/password/reset/").parametrize()

@schema.given(data=st.data())

def test\_password\_reset(data, case, user):

if data.draw(st.booleans()):

case.body["token"] = data.draw(

(st.emails() | st.just(user.email)).map(create\_reset\_password\_token)

)

case.call\_and\_validate()

NOTE:

Обратите внимание, что тесты, использующие пользовательские примеры Hypothesis, не будут работать, если ваша схема содержит явные примеры. Они несовместимы, поскольку Schemathesis только создает аргумент case из примеров и не знает, какие значения предоставить для других аргументов, которые вы определяете для своей тестовой функции.

Помните о ключевом ограничении при интеграции Schemathesis с Hypothesis и pytest для тестирования. Schemathesis не может одновременно поддерживать пользовательские стратегии Hypothesis и явные примеры, определенные в вашей схеме API. Это ограничение возникает из-за того, что Schemathesis генерирует экземпляры hypothesis.example из примеров, определенных схемой, но у него нет возможности выводить или назначать соответствующие значения для дополнительных пользовательских аргументов в ваших тестовых функциях. Чтобы эффективно управлять этим, вам следует рассмотреть возможность структурирования ваших тестов по-другому. Для тестов, включающих пользовательские стратегии Hypothesis, вам необходимо исключить Phase.explicit, чтобы избежать конфликтов.

### Настройка валидации

В новой версии правильно делать через schemathesis.toml

[checks]

enabled = false

# Only check for server errors and schema compliance

not\_a\_server\_error.enabled = true

response\_schema\_conformance.enabled = true

НЕ ПОЛУЧИЛОСЬ ОТКЛЮЧИТЬ not\_a\_server\_error:

[checks]

not\_a\_server\_error.enabled = false

status\_code\_conformance.enabled = false

content\_type\_conformance.enabled = false

response\_schema\_conformance.enabled = false

positive\_data\_acceptance.enabled = false

negative\_data\_rejection.enabled = false

use\_after\_free.enabled = false

ensure\_resource\_availability.enabled = false

missing\_required\_header.enabled = false

ignored\_auth.enabled = false

unsupported\_method.enabled = false

### Настройки проведения тестов

Поскольку после обработки схемы тесты эндпоинтов – это обычные тесты hypothesis, то с ними можно использовать декоратор hypothesis.settings и все другие настройки hypothesis.

<https://hypothesis.readthedocs.io/en/latest/reference/api.html#settings>

*Локально*

Указать количество генерируемых семплов для конкретного теста:

@schema.parametrize()

@settings(max\_examples=10)

def test\_app(case):

    case.call\_and\_validate()

*Глобально*

В начале

settings.register\_profile( "ci", settings.get\_profile("ci"), max\_examples=1000, )

### Фичи

1) Подставление нужных значений в конкретные параметры эндпоинтов:

@schema.parametrize()

@schema.override(path\_parameters={"user\_id": "42"}, query={"apiKey": "secret"})

def test\_api(case):

2) Повторное использование ошибок:

Schemathesis сохраняет неудачные тестовые случаи в кэше (каталог .hypothesis) и повторно запускает их в последующих запусках для обнаружения регрессий. Эта фаза может сократить общее количество сгенерированных тестовых случаев, поскольку Schemathesis может обнаружить сбои раньше.

3) Explicit examples

Если в схеме есть прямые примеры, то schemathesis протестирует сначала их, подставляя сгенерированные значения в пустые параметры.

*# схема*

{

"type": "object",

"properties": {

"name": {"type": "string", "example": "John"},

"age": {"type": "integer", "examples": [42, 43]},

"street": {"type": "string"}

}

}

*# Test cases*

{"name": "John", "age": 42, "street": "<RANDOM STRING>"}

{"name": "John", "age": 43, "street": "<ANOTHER RANDOM STRING>"}

4) Ограничение скорости

API реализуют ограничение скорости для предотвращения нецелевого использования своих ресурсов. Загрузчики схем принимают аргумент rate\_limit, который можно использовать для установки максимального количества запросов в секунду, минуту, час или день во время тестирования, чтобы избежать достижения этих ограничений.

import schemathesis

*# 3 requests per second - `3/s`*

*# 100 requests per minute - `100/m`*

*# 1000 requests per hour - `1000/h`*

*# 10000 requests per day - `10000/d`*

RATE\_LIMIT = "3/s"

schema = schemathesis.from\_uri(

"https://example.schemathesis.io/openapi.json",

rate\_limit=RATE\_LIMIT,

)

### Логирование

Создание test\_logs.txt в котором скопирован лог из консоли

pytest test\_api.py > test\_logs.txt 2>&1

Создаие junit.xml

Формат в котором тольо основная часть логов

pytest --junitxml=./test.xml

### \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### Проверка фукционала schemathesis

#### Генерация данных

#### Валидация

Для проверки создан простой сервис fastapi