API TESTING WITH PASSION: APPROACHES AND PYTHON TOOLKIT

IGOR BALAGUROV

QA AUTOMATION TECH LEAD AT VK

- vk.com/korgan
- linkedin.com/in/ibalagurov
- github.com/ibalagurov

TALKS

- Heisenbug 2018 Piter: Don’t repeat yourself: UI-тесты для Веб, iOS и Android одновременно
- Heisenbug 2020 Piter: API автотесты: Чек-лист с примерами на Python
- QA Guild Podcast: Разговоры про Python

VK EXPERIENCE

- VK Fin-tech
  - Wallets (VK Pay)
- Internal Development
  - A lot of small internal services

PREREQUISITE

- Processes
- Exploratory testing
- Base acceptance autotests
- Some bugs in production
- …

PREREQUISITE

- Processes
- Exploratory testing
- Base acceptance autotests
- Some bugs in production
- ...
- Find more bugs automatically
- Reduce time to market

COMMON APPROACH

- Everybody knows about the test pyramid
- Flaky tests are almost defeated
- Evergreen autotests:
  - No failures
  - No bugs

COMMON APPROACH

- Everybody knows about the test pyramid
- Flaky tests are almost defeated
- Evergreen autotests:
  - No failures
  - No bugs
    - The product is dead?
    - Autotests are useless?

PLAN

- Problem
- Approach
- Tools
  - Examples
- Pros & Cons
- What’s next?
- Conclusions

PROBLEM: HIDDEN CORNER CASES

- Pesticide Paradox:
  - Hardcode values in test data
  - Inability to variate data across all tests

APPROACH

- Randomization in getting boundary values and inside equivalence classes
- Randomization helps to test:
  - Combinations
  - Localization
  - New checks
- Seed for reproducible tests

TOOLS

- Mimezis
- Faker
- Factoryboy

EXAMPLES: POSITIVE CASES

- Boundary values
- Equivalence classes

EXAMPLES: POSITIVE CASES

```python
import mimesis

generate = mimesis.Generic(locale="en", seed=100500)

currency = generate.business.currency_iso_code()
# 'USD'

phone = generate.person.telephone()
# '1-115-514-0222'
```

EXAMPLES: POSITIVE CASES

```
1 import mimesis
2 from mimesis.random import Random
3
4 generate = mimesis.Generic(locale="en", seed=100500)
5 random = Random(x=100500)
6
7 number = generate.numbers.integer_number
8 choice = generate.choice
```

EXAMPLES: POSITIVE CASES

```
1 # boundary values
2 max_value = ["10000", "10000.0", "10000.00"]
3 min_value = ["1", "1.0", "1.00"]
4 boundary_value = choice(min_value + max_value)
5 # '10000.0'
```

EXAMPLES: POSITIVE CASES

```
# equivalence classes
decimal_classes = ["{}", "{}#", "{}##"]
fractional_classes = ["", ".#", ".##"]
decimal_template = choice(decimal_classes)
# '{}#'
non_zero_digit = number(start=1, end=9)
# 9
decimal_part = decimal_template.format(non_zero_digit)
fractional_part = choice(fractional_classes)
# ''
mask = decimal_part + fractional_part
equivalence_class = random.custom_code(mask=mask)
# '95'
```

EXAMPLES: POSITIVE CASES

```
def valid_amount():
    ...
    result = choice([boundary_value, equivalence_class])
    return result

valid_param_dict = {
    "amount": valid_amount()
}

def generate_valid(param):
    return valid_param_dict.get(param)
```

EXAMPLES: NEGATIVE CASES

- Nulls
- Invalid format values
- Non-existent values

EXAMPLES: NEGATIVE CASES

```python
import mimesis

generate = mimesis.Generic(locale="en", seed=100500)

nulls = ["null", "nan", "nil", "none"]
random_word = generate.text.word()
common = [random_word, *nulls]

invalid_currency = generate.choice(common)
# "none"
```

EXAMPLES: NEGATIVE CASES

```
invalid_param_dict = {
    "currency": generate.choice(common)
}


def generate_invalid(param):
    return invalid_param_dict.get(param)
```

EXAMPLES: NEGATIVE CASES

```
1 from test_data import generate_invalid
2
3 default_payment_data = {"amount": 1, "currency": "RUB"}
4
5 data = {
6     **default_payment_data,
7     "currency": generate_invalid("currency")
8 }
```

EXAMPLES: NEGATIVE CASES

```
param = "currency"

data = {
    **default_payment_data,
    param: generate_invalid(param)
}
```

EXAMPLES: PARAMETRIZATION

EXAMPLES: PARAMETRIZATION

```
1 import pytest
2 import requests
3 from test_data import generate_invalid
```

EXAMPLES: PARAMETRIZATION

```
default_payment_data = {"amount": 1, "currency": "RUB"}

@pytest.mark.parametrize(
    "param", default_payment_data.keys()
)
def test_payment_invalid_param(param):
    data = {
        **default_payment_data,
        param: generate_invalid(param)
    }

    requests.post(url="/payment", data=data)
```

RANDOMIZATION & PARAMETRIZATION: PROS & CONS

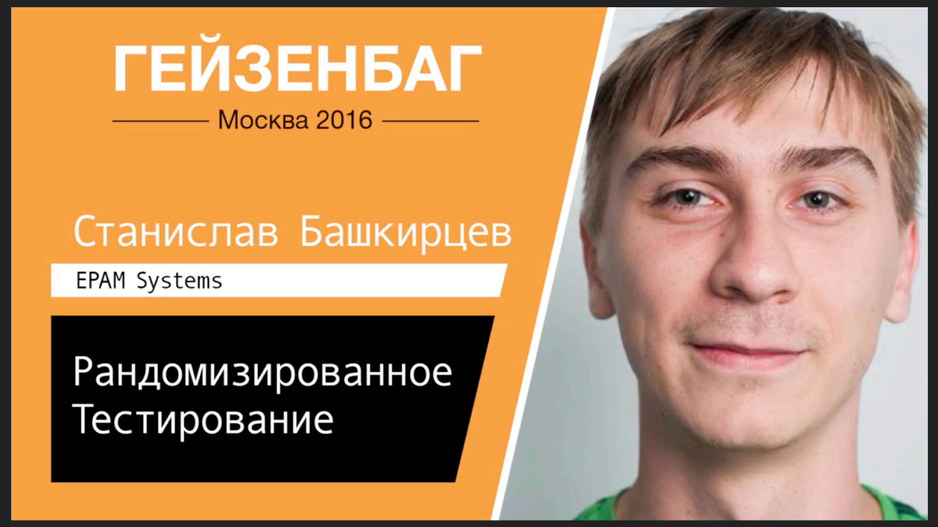
- No routine
- Reusable values
  - Common for same or similar parameters
  - It could be used as standalone data generator solution
- Combinations
- ...

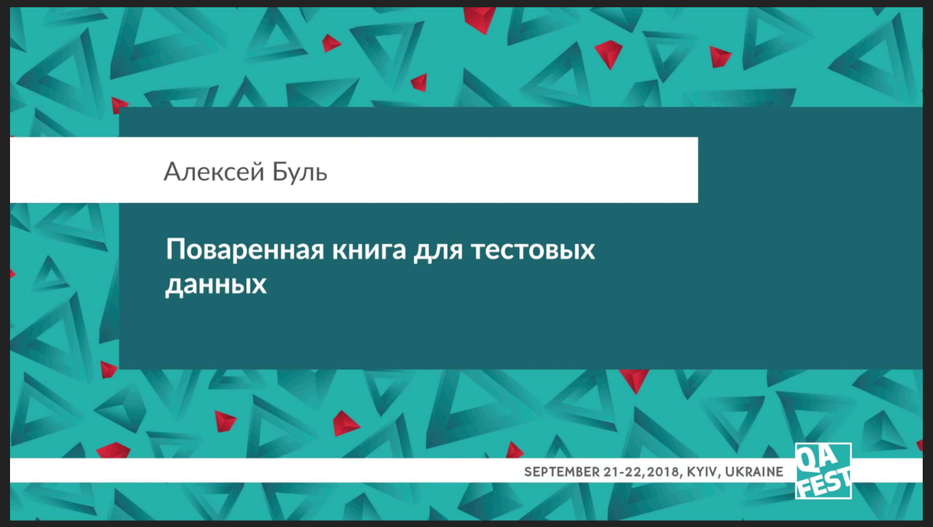
RANDOMIZATION & PARAMETRIZATION: PROS & CONS

- No routine
- Reusable values:
  - Common for same parameters
  - It could be used as standalone data generator solution
- Combinations
- Sometimes tests fail → Knowledge mining
- You need the seed to reproduce test

WHAT'S NEXT?

- Mimesis documentation
- Articles (habr)
- Heisenbug 2016 Moscow: Randomized testing
- QA Fest 2018: Поваренная книга для тестовых данных



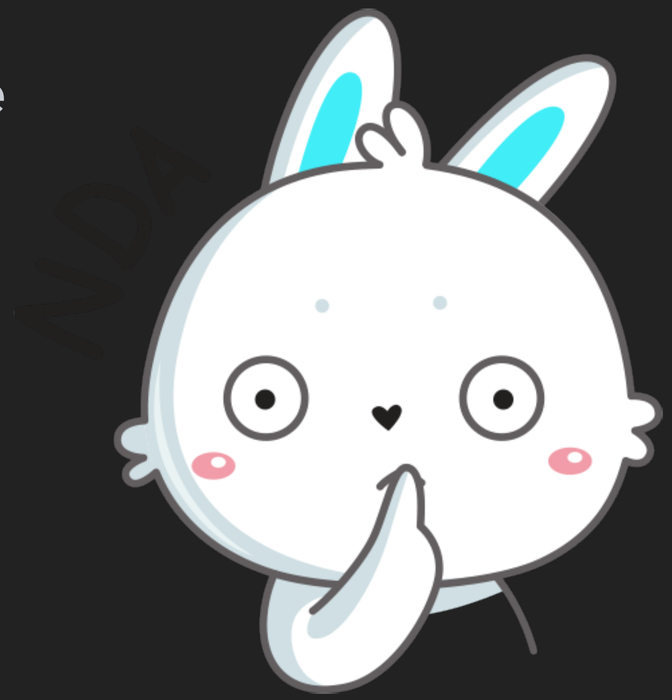


PROBLEM: COMPLEX DATA VALIDATION

- 3rd-parties data
- Sensitive data:
  - Any new field is a risk
- Contract:
  - Any new format type is a risk

APPROACH

- Data validation, schemas
  - Body contains only predefined fields, in appropriate format
  - No additional information: in any case, in any place
  - …
  - There is no any sensitive data



TOOLS

- Cerberus, Pydantic, jsonschema
  - Convenient API: no need in additional conversion
  - All errors in one check
  - Required / Unkown fields
  - Custom rules (any, all, none, dependencies)

EXAMPLES

```
 1 from cerberus import Validator
 2
 3 schema = {'name': {'type': 'string'}}
 4
 5 v = Validator(schema)
 6
 7 document = {'name': 'john doe'}
 8
 9 v.validate(document)
10 # True
```

EXAMPLES

```python
from src import vk_pay, check
from src import data_service


def test_wallet_balance():
    wallet = data_service.get_wallet()

    response = vk_pay.wallet_balance(wallet=wallet)
    response.json()
    # {"data": {"userId": 1, "balance": 100}}

    check.response_matches_ok_schema(response)
```

EXAMPLES

```
wallet_balance = {
    "data": {
        "type": "dict",
        "required": True,
        "require_all": True,
        "schema": {
            "balance": {
                "type": "number", "min": 0, "max": 60000
            },
            "userId": {"type": "integer", "min": 1},
        }
    }
}
```

EXAMPLES

```
ok_mapping = {
    "get": {
        "/wallet/balance": wallet_balance,
    }
}

def for_ok_schema(response):
    url = response.request.path_url
    method = response.request.method.lower()
    return ok_mapping.get(method, {}).get(url)
```

EXAMPLES

```
 1 from src import schema
 2 from cerberus import Validator
 3
 4 def response_matches_ok_schema(response):
 5     response_schema = schema.for_ok_response(response)
 6     data_to_validate = response.json()
 7     schema_validator = Validator(response_schema)
 8
 9     assert schema_validator.validate(data_to_validate), \
10         schema_validator.errors
```

EXAMPLES

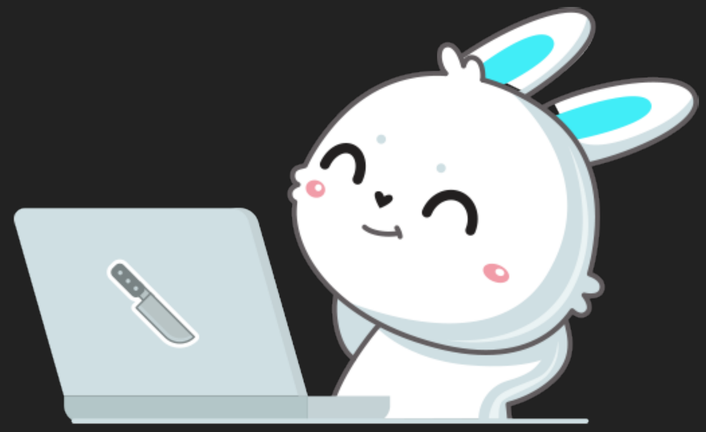
```
data_to_validate = response.json()
# {"data": {"userId": 0, "balance": -0.1}}

schema_validator.validate(data_to_validate))
# False

schema_validator.errors
# {'data':[{
#     'balance': ['min value is 0'],
#     'userId': ['min value is 1']
# }]}
```

DATA VALIDATION: PROS & CONS

- Useful with 3rd parties
- If you want to know about:
  - Every new field
  - Any new format
- …

DATA VALIDATION: PROS & CONS

- Useful with 3rd parties
- If you want to know about:
  - Every new field
  - Any new format
- …
- Sometimes tests fail → Knowledge mining
- You need to describe your schemas



WHAT'S NEXT?

- PiterPy 2018: Cerberus, or Data Validation for Humans
- Cerberus documentation

PROBLEM: BASE SECURITY TESTS

- Security risks
- Sensitive data
- Separate security department:
  - It is not so iterative, as it could be

APPROACH

- OWASP API Security:
  - Privacy / Permissions
  - Wrong tokens / hashes / signs / private keys
  - Invalid values (overflow attempts, injections)
  - Rate & Resource limits (flood control)
  - Race conditions

APPROACH

- Negative actors → privacy
- Negative values → resources limits
- Wrong headers → tokens / hashes
- …
- It’s not a problem to extend tests

APPROACH

- Check every request param
  - Overflow attempts
  - Injections
- Check concurrent responses
  - Rate limits
  - Rece conditions

EXAMPLES: NEGATIVE CASES

- Nulls
- Invalid format values
- Non-existent values

EXAMPLES: NEGATIVE CASES

- Nulls
- Invalid format values
- Non-existent values

- Overflow attempts
- Injections

EXAMPLES: NEGATIVE CASES

```python
import mimesis

generate = mimesis.Generic(locale="en", seed=100500)

nulls = ["null", "nan", "nil", "none"]
random_word = generate.text.word()
common = [random_word, *nulls]

invalid_currency = generate.choice(common)
# "none"
```

EXAMPLES: NEGATIVE CASES

```
1 overflow_int = "9" * 100
2
3 my_cool_injections = ["' OR 1 == 1"]
4
```

EXAMPLES: NEGATIVE CASES

```python
import mimesis

generate = mimesis.Generic(locale="en", seed=100500)

nulls = ["null", "nan", "nil", "none"]
my_cool_injections = ["' OR 1 == 1"]
overflow_int = "9" * 100
common = [overflow_int, *nulls, *my_cool_injections]

invalid_currency = generate.choice(common)
# "none"
```

APPROACH

- Check every request param
  - Overflow attempts
  - Injections
- Check concurrent responses
  - Rate limits
  - Rece conditions

APPROACH

- Check concurrent responses
  - Rate limits
  - Rece conditions

TOOLS

- Standard library:
  - concurrent.futures
  - threading
  - asyncio
  - and etc.

EXAMPLES

```python
from functools import partial
from src import vk_pay, test_data, concurrent

transaction_id = test_data.create_transaction()

api_method = partial(
    func=vk_pay.confirm_transaction,
    transaction_id=transaction_id
)
```

EXAMPLES

```
def test_rate_limits():
    responses = concurrent.execute(
        func=api_method, pause_between=0.1, times=10
    )

    assert any(
        response.status_code == 429
        for response in responses
    )
```

EXAMPLES

```
def test_race_condition():
    responses = concurrent.execute(
        func=api_method, times=10
    )

    ok_responses = [
        response.status_code == 200
        for response in responses
    ]

    assert len(ok_responses) == 1
```

EXAMPLES

```
def test_small_stress():
    responses = concurrent.execute(
        func=api_method, times=100
    )

    long_responses = [
        response for response in responses
        if response.elapsed.total_seconds() >= 2
    ]

    assert not long_responses
```

EXAMPLES

```
1 import time
2 from concurrent.futures import ThreadPoolExecutor
3
4 def execute(func, times=1, pause_between=None):
5     pool = ThreadPoolExecutor(max_workers=times)
```

EXAMPLES

```
if pause_between is None:
     futures = [
         pool.submit(func) for _ in range(times)
     ]

     return [f.result() for f in futures]
```

EXAMPLES

```
else:
    futures = []
    for _ in range(times - 1):
        futures.append(pool.submit(func))
        time.sleep(pause_between)
    futures.append(pool.submit(func))

    return [f.result() for f in futures]
```

EXAMPLES

```
if pause_between is None:
    futures = [
        pool.submit(func) for _ in range(times)
    ]
else:
    futures = []
    for _ in range(times - 1):
        futures.append(pool.submit(func))
        time.sleep(pause_between)
    futures.append(pool.submit(func))

return [f.result() for f in futures]
```

PROS & CONS

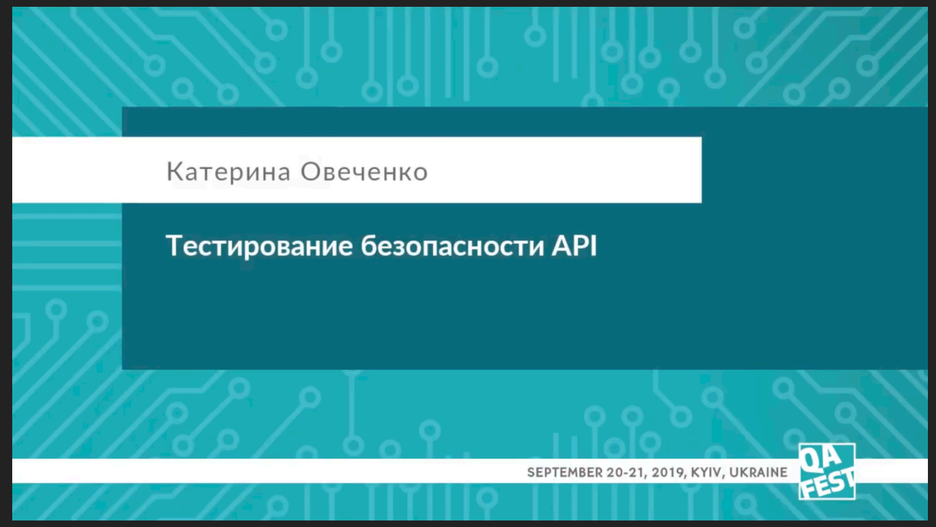
- If you want to prevent or find earlier:
  - Security issues
  - Performance issues
- …

PROS & CONS

- If you want to prevent or find earlier:
  - Security issues
  - Performance issues
- …
- Sometimes tests fail → Knowledge mining
- Learning and investigation time

WHAT'S NEXT?

- OWASP API Security
- QA Fest 2019: Security API Testing



PROBLEM: A LOT OF MANUALLY MADE AUTOTESTS

- Every new service require much time

APPROACH

- API Schema (Swagger, GraphQL, Something custom*)
- Parsing
- Generate data for each type
- Fuzzing (send requests)
  - Each method
  - Each field
  - Check responses

APPROACH

- Parsing schema
- Property-based testing

TOOLS

- Swagger
- Hypothesis
- swagger-conformance (pyswagger + hypothesis)
- hypothesis-jsonschema (jsonschema + hypothesis)

TOOLS

- Schemathesis (hypothesis-jsonschema)
  - Swagger 2.0 / OpenAPI 3.0
  - GraphQL
    - Full-featured support - in progress
  - Requests
  - Pytest (parametrization)

EXAMPLES: CONNECT TO SWAGGER SCHEMA

```python
import schemathesis

schema = schemathesis.from_uri(
    "https://my-cool-service.com/v2/api-docs",
    verify=False
)


@schema.parametrize()
def test_no_server_errors(case):
    ...
```

EXAMPLES: QUICK START FROM DOCUMENTATION

```python
@schema.parametrize()
def test_no_server_errors(case):
    # `requests` will make an appropriate call
    response = case.call(verify=False)
    # You could use built-in checks
    # case.validate_response(response)
    # Or assert the response manually
    assert response.status_code < 500
```

EXAMPLES: REAL WORLD

```python
@pytest.mark.parametrize("user_role", [...])
@schema.parametrize()
def test_no_server_errors(case, arrange_user, user_role):
    user = arrange_user(role=user_role)
    client = http_client.auth_session(user=user)

    response = case.call(
        headers=client.headers, cookies=client.cookies
    )

    assert response.status_code < 500, "Server error"
    assert response.status_code != 401, "Auth problems"
```

EXAMPLES: PROPERTY CONFIGURATION

```
from hypothesis import HealthCheck

@hypothesis.settings(
    suppress_health_check=HealthCheck.all(),
    max_examples=config.FUZZING_MAX_EXAMPLES
)
def test_no_server_errors(...):
    ...
```

EXAMPLES: CHECK AUTH

```
1 # Negative Lookahead
2 methods_with_auth_regex = r"^(?!/api/without_auth/).*$"
3
4
5 @hypothesis.settings(max_examples=1)
6 @schema.parametrize(endpoint=methods_with_auth_regex)
7 def test_without_auth_cookies(case):
8     ...
```

EXAMPLES: CHECK AUTH

```
cookies = http_client.auth_session(...).cookies
cookies_without_session = {
    k: v for k, v in cookies.items() if k != "session"
}

response = case.call(
    headers=headers, cookies=cookies_without_session
)

assert response.status_code in [401, 404]
assert "results" not in response.text
```

EXAMPLES: FILTERING METHODS

```python
methods_with_auth_regex = r"^(?!/api/without_auth/).*$"
not_get_api_methods = ["post", "put", "patch", "delete"]


@hypothesis.settings(max_examples=1)
@schema.parametrize(
    endpoint=methods_with_auth_regex,
    method=not_get_api_methods
)
def test_some_logic(case, arrange_user):
    ...
```

EXAMPLES: BUILT IN VALIDATION RULES

```
1 @pytest.mark.parametrize("user_role", [...])
2 @schema.parametrize()
3 def test_no_server_errors(case, arrange_user, user_role):
4     user = arrange_user(..)
5     client = http_client.auth_session(...)
6
7     response = case.call(...)
8
9     case.validate_response(response=response)
```

EXAMPLES: EXTEND VALIDATION RULES

```
 1 @pytest.mark.parametrize("user_role", [...])
 2 @schema.parametrize()
 3 def test_no_server_errors(case, arrange_user, user_role):
 4     user = arrange_user(..)
 5     client = http_client.auth_session(...)
 6
 7     response = case.call(...)
 8
 9     fuzzing.validate_response(
10         response=response, case=case
11     )
```

EXAMPLES: EXTEND VALIDATION RULES

```python
from schemathesis.models import Case


def response_time_check(
        response: Response, _case: Case = None
) -> None:
    """Check response time"""
    actual_time = response.elapsed.total_seconds()
    expected_time = 1.0
    assert (
        actual_time <= expected_time
    ), f"Expected: {expected_time}\nActual:{actual_time}"
```

EXAMPLES: EXTEND VALIDATION RULES

```python
from schemathesis.checks import
response_schema_conformance

def response_time_check(...) -> None:
    ...

def validate_response(response: Response, case: Case):
    checks = (
        response_schema_conformance, response_time_check
    )
    case.validate_response(
        response=response, checks=checks
    )
```

EXAMPLES: EXTEND VALIDATION RULES

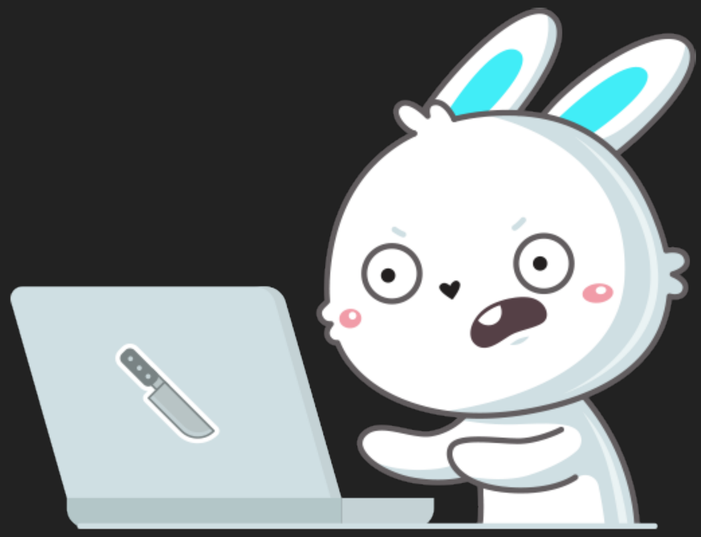
```python
@pytest.mark.parametrize("user_role", [...])
@schema.parametrize()
def test_no_server_errors(case, arrange_user, user_role):
    user = arrange_user(..)
    client = http_client.auth_session(...)

    response = case.call(...)

    fuzzing.validate_response(
        response=response, case=case
    )
```

PROS

- Rapid start testing for new services:
  - Schema → few minutes → several 5xx errors
- Declarative approach:
  - Describe rules that should be validated
- Lazy test automation:
  - No need to write tests on every new API method

CONS

- Send all required fields and data that fits types
- Need to suppress warnings from Hypothesis
- Require up-to-date swagger schema
  - Could fail on schema parsing
- Pytest-xdist parallel issues
- Many layers → issues require time
- Many dependencies

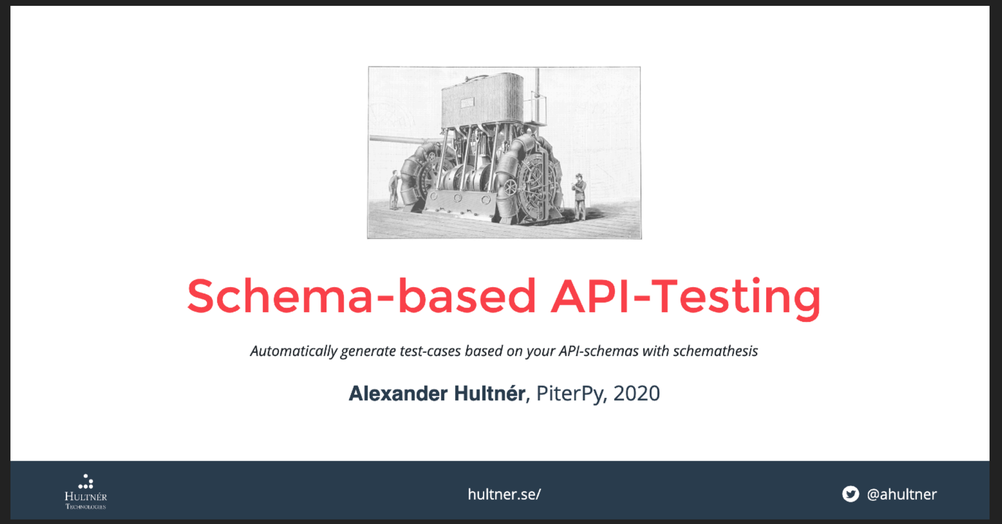


WHAT'S NEXT?

- Introduction to property-based testing (Hypothesis)
- Choosing properties for property-based testing (F#)
- Hypothesis documentation
- Heisenbug 2019 Piter: Property testing: Strategic automation for devs and SDETs

WHAT'S NEXT?

- PiterPy 2020: Schema-based API-Testing – Automatically generate test-cases with schemathesis
- Progress report and plan (2020-10-05)
- Habr (Russian)
- Medium (English)

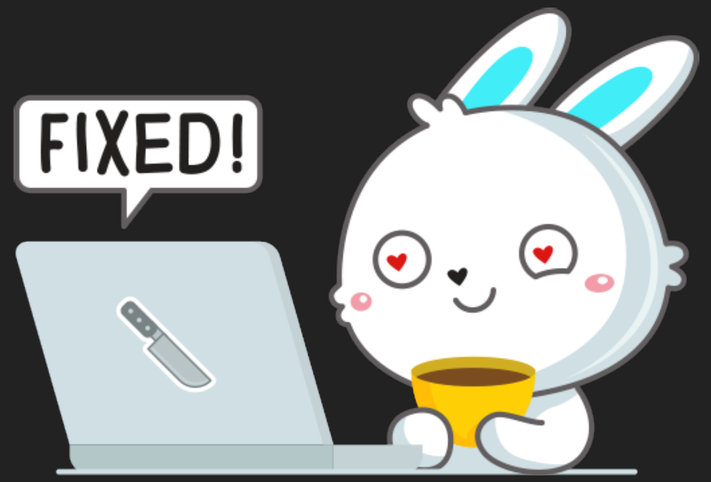


REMEMBER ALL

- Approaches & Tools & Examples:
  - Data generation (Mimesis)
  - Data validation (Cerberus)
  - Concurrent responses (Concurrent.futures)
  - API Fuzzing (Schemathesis)

LET'S TEST THE NEW API

- API Fuzzing (Schemathesis)
- Race conditions, rate limits (Concurrent.futures)
- Data generation (Mimesis)
- Data validation (Cerberus)



WHAT'S IT ABOUT?

- Test more → Prevent more
- Explore your system → Knowledge mining
- Make more general solutions → Automate & Chill
- Keep think like QA → Automate like a Boss
- …

WHAT'S IT ABOUT?

- Test more → Prevent more
- Explore your system → Knowledge mining
- Make more general solutions → Automate & Chill
- Keep think like QA → Automate like a Boss
- ...
- Learn hard

WHAT'S NEXT

- API schemas + Property-base testing

THANK YOU!!!

vk.com

presentation

linkedin.com