REST API Automation With Python 101

Whoami – Oleg Nikiforov

14 years in Mobile and Web testing, automation and manual

8 years in Lohika / Capgemini Engineering

Numerous (50+) projects for Android, iOS, IoT and web

Passionate about API testing and solutions architecture

Speaker on international conferences

Play Padel in free time

Agenda

What is a RESTful API

Test Pyramid

Why we need API testing

Pros / Cons of API tests

API vs UI tests

Classic (unittest) vs pytest approaches

requests module overview

Allure — why reporting matters

Validation: jsonschema vs dataclass vs pydantic

Setting an HTTP proxy for debugging

AI in API test writing

What is a RESTful API - Taiga.io

Taiga.io - Free and open-source agile project management platform

- Tracks *projects, user stories, tasks, issues, sprints*

- Public instance: https://api.taiga.io/api/v1

- Well-documented REST API (a bit outdated though)

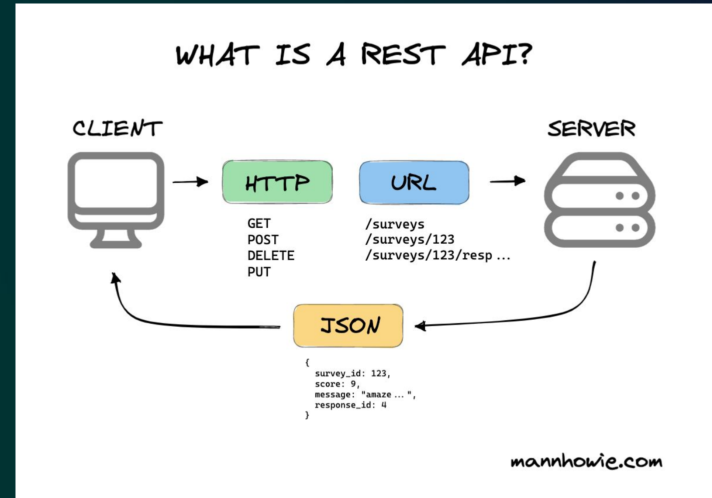
What we use in this workshop:

Resource	Endpoint	Why
Auth token	POST /auth	Login, get Bearer token
User Stories	GET/POST/PATCH/DELETE /userstories	Full CRUD lifecycle
Project	GET /projects/{id}	Read project metadata

What is a RESTful API

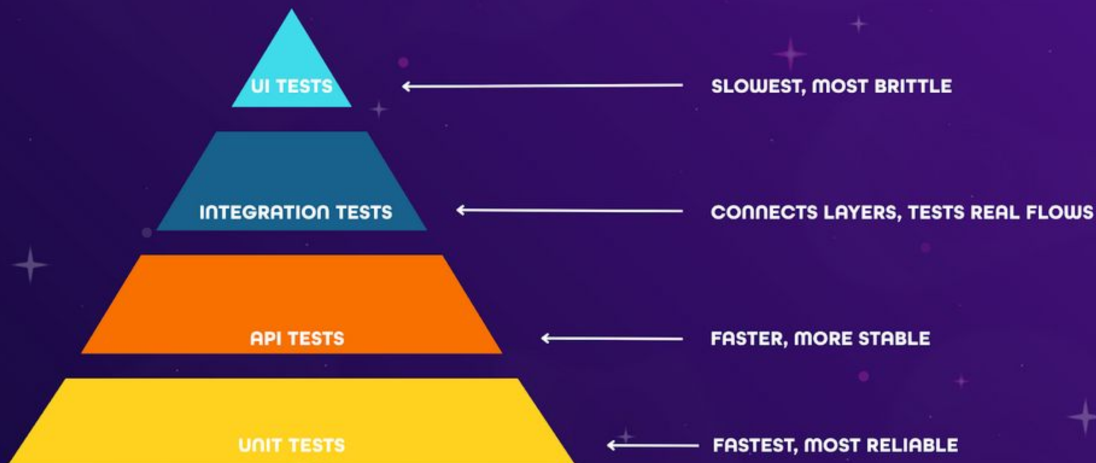
REpresentational State Transfer over HTTP

- Stateless — server holds no session between calls
- Resource-oriented — URLs name things, not actions
- JSON — the universal payload format



HTTP Verb	CRUD	Taiga example
GET	Read	/userstories?project=ID
POST	Create	/userstories
PUT	Update	/userstories/ID
DELETE	Delete	/userstories/ID

Testing Pyramid



UI (E2E):

few · slow · expensive

API/Integration:

some · medium · stable

Unit:

many · fast · cheap

The place of API tests in the ~~food~~ tests chain — and Why

Filling the gap:

- Unit tests verify *internal logic* — and not the fact that services interact correctly
- UI / E2E tests verify *user journeys* — but are slow and flaky
- **API** tests verify *contracts between systems* — fast, stable, do not use browser

Why

- Microservices and separate frontend/backend teams made API contracts the primary seam
- A broken JSON field can silently corrupt the UI — API tests are able to catch the issue first
- Parallel execution: 100 API tests can finish before a single Cypress suite loads

Pros and Cons of API Tests

	API tests	UI tests (Playwright / Cypress / Selenium, etc.)
Speed	ms / seconds per test	seconds / minutes per test
Flakiness	low (depends on the backend stability and test code quality)	higher (animation, timing, selectors)
Coverage	HTTP contract, data, auth	full user journey, rendering, UX
Feedback time	fast (CI < 1 min)	slow (CI 10-30 min)
Debug cost	log the request/response	screenshot + video + DOM
When to prefer	Contract (endpoints), auth, data validation	user flows, visual regression

Choose the right tool for the layer. They complement, not replace, each other.

A "good" API test - characteristics

- Idempotent - multiple runs of the same test doesn't change system's state
- Isolated / independet - can run on it's own and creates it's own data (use randomization during test collection)
- Makes the right assertions - not only status code
- Covers not only happy path - this rather applies to the test suite in general
- Quick - relatively quick (not tens of seconds) to not change the regular test run into "nightly"

A "good" API test - anatomy

The AAA pattern - Arrange, Act, Assert

VS

AAAT - Arrange, Act, Assert, Teardown

- Arrange / Setup - actions to create needed data / alter settings: API calls, DB interactions
- Act / Test - executing the requests under test
- Assert - proper assertions, usually more than 1, as opposed to classic AAA approach
- Teardown - actions to "reverse" the setup - delete created data, revert the settings changes: API calls, DB interactions

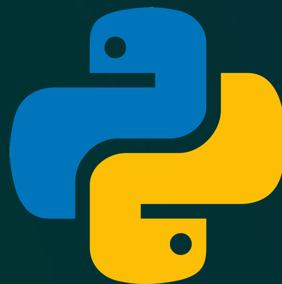
Before we start practical part

Python

unittest vs Pytest approach

requests module

Allure reporting



Branches:

main

step1-basic-tests

Python

Python is interpreted — no compile step, runs line by line at runtime.

Type checking: Runtime (optional static via *mypy*)

Performance: Slower CPU-bound tasks

Syntax: Minimal boilerplate

Why Python is good for API testing?

- Write a test, run it immediately — no build pipeline
- Vast ecosystem: `pytest`, `requests`, `pydantic` all install in seconds
- Readable assertions — test code reads like prose

unittest vs Pytest

Concern	unittest (classical OOP)	pytest (functional/fixture)
Mental model	Inheritance hierarchy	Dependency injection
Setup scope	Method on class (setUp)	Fixture with scope= param
Teardown	tearDown method	yield in same fixture function
Shared state across tests	setUpClass / tearDownClass	scope="class" or scope="module"
Session-wide setup	Not native (need addModuleCleanup)	scope="session" fixture
Reuse across files	Inheritance from base class	conftest.py — auto-discovered
Parameterization	subTest()	@pytest.mark.parametrize
Assertions	assertEqual, assertIn, etc.	Plain assert

requests

requests is considered as mostly used http client in Python

It's built on top of *urllib3* engine

Created: February 13, 2011 (v0.0.1)

- Current version: 2.34.2
- Released: May 14, 2026

Key concepts that we'll use today:

- `requests.Session()` — shared headers, cookies, connection pool
- `session.hooks["response"]` — run callbacks on every response (logging)
- `log_response_hook` in *conftest.py* — strips `Authorization` header before logging
- `timeout=` on every call — never omit it

Syntax

Assuming our base URL is "https://api.taiga.io/api/v1", a GET request to obtain project details will look like this:

```
import requests
requests.get(
    url="https://api.taiga.io/api/v1/projects",
    headers='{"Authorization": f"Bearer {token_value}'}
)
```

This returns a *Response* object containing all the data about both request and response

Allure or why reporting matters

A passing test suite no one can read is as useless as no tests at all.

What good reporting enables:

- Shared status — the same report artifact is shared across the team and stakeholders if necessary
- Failure debugging — request body + response + step context visible in the report (screenshots and videos for UI tests)
- Trend detection — new or recurring failures visible in the history
- Handoff — reports can be used a source of information for the knowledge transfer

Features:

- Collapsible steps in the report UI
- Attachments (request / response body, screenshots, videos)
- History graph — see trend across runs
- @allure.feature / @allure.story for grouping

“TDD” for test automation

It is important to understand that assertions in the tests we write actually work

In regular TDD (test driven development) the unit test is written first, it fails because there is no implementation. Next implementation is added and the test should pass. If it doesn’t pass - implementation is refactored until the test passes.

In test automation, the test is written, and we should flip assertions to fail and make sure that they actually assert for the right thing. Then we flip them back and make sure that the test succeeds.

```
assert resp_data["id"] == self.cfg.project_id
>
assert resp_data["id"] != self.cfg.project_id
```

1st test

Let’s write our first test to get the project data

The flow is next:

1. Login, obtain token
2. Make a request to GET /projects/project_id
3. assert status code is 200

Any other assertions?

Checkout branch “main”

Create *tests* directory

Create *test_project_basic.py* file in *tests* dir

Adding fixtures

Fixtures work as dependency injection - usually you don't call them directly.

conftest.py - main file for fixtures, unless you have a big number and need to split them across files

yield - splits the fixture's execution into two phases:

1. Setup phase (before yield)
2. Cleanup phase (after yield)

Example: `user_story` fixture

Fixture scopes

Fixtures are created when first requested by a test, and are destroyed based on their scope:

function: the default scope, the fixture is destroyed at the end of the test.

class: the fixture is destroyed during teardown of the last test in the class.

module: the fixture is destroyed during teardown of the last test in the module.

package: the fixture is destroyed during teardown of the last test in the package.

session: the fixture is destroyed at the end of the test session.

Today we will be using *session* and *function* scopes

Response Validation — Three Styles

Why should we validate response structure at all?

- Status 200 ≠ correct payload
- A field renamed server-side silently breaks the consumer (client)
- Without validation, your tests assert *reachability*, not *contract* validity

Three approaches that we use in this repo:

Approach	File	What it checks
jsonschema	schemas/user_story.schema.json	Shape + types (JSON contract)
dataclass	tests/models/dataclasses/	Presence of required fields
pydantic	tests/models/pydantic/	Types + clear errors

jsonschema — Contract-First

`schemas/user_story.schema.json + tests/step3_validation/test_user_stories_jsonschema.py`

Usage:

```
jsonschema.validate(instance=resp.json(), schema=user_story_schema)
```

Strengths:

- Schema lives in a `.json` file — shareable with any language / team
- Validates shape and types without touching Python models

Limitations:

- No Python type hints — schema and code drift independently
- Error messages can be verbose and hard to parse
- No coercion — if API returns "7" where *int* expected, it fails

dataclass vs pydantic

	dataclass	pydantic
Source	stdlib	third-party
Validation	manual (*from_dict*)	automatic (*model_validate*)
Error messages	whatever you raise	rich, field-level
Extra fields	crashes or ignored by *from_dict*	extra="ignore"

```
tests/models/dataclasses/user_story_dataclass.py:
story = UserStory(**resp.json())          # crashes on unexpected fields
story = UserStory.from_dict(resp.json())  # guarded: raises ValueError
```

```
tests/models/pydantic/user_story_pydantic_model.py: story = UserStory.model_validate(resp.json())
```

HTTP Proxy for Debugging

Use per-session setting:

```
session.proxies = {"https": "http://localhost:8888"}
session.verify = False # This is important to bypass cert validation error for self-hosted certificates #
only when you trust the proxy CA
```

What you gain:

- See exact request/response data in a readable format

- Compare what requests module sends vs what the server returns

- Rapidly spot encoding, header, or body serialisation bugs

AI's Role in API Test Writing

What AI does well:

- Generate boilerplate from an OpenAPI spec or curl example
- Suggest `@pytest.mark.parametrize` matrices for edge cases
- Draft schema files from a sample response JSON

What AI gets wrong:

- Intent — it doesn't know what matters to validate
- The oracle problem — it can't know what the correct value should be
- Hallucinated endpoints — always verify against real API docs
- Test isolation — may suggest shared state that causes flaky tests
- Test pile - AI tends to create a huge amount of tests even though half of those won't be really needed

Rule: AI makes the drafts, and QA Engineer reviews. Never should it be the other way around.

Conclusion

Key takeaways:

1. Know which layer you're testing and why. Use the testing pyramid as a reference
2. Validation can be extensive. Checking only status code is almost never enough, validating the structure is extremely important
3. Find the approach that works for you: requests vs https, pytest vs unittest, brief vs detailed assertions, jsonschema vs dataclass vs pydantic - choose the tool that you are comfortable with and that gives you the best result. There is no “silver bullet” in automation testing.

Q&A + References

Useful links:

https://github.com/onikiforov/NTD-python-101

https://docs.pytest.org/en/stable/

https://docs.pytest.org/en/stable/explanation/fixtures.html

https://requests.readthedocs.io/en/latest/

https://allurereport.org/docs/

https://dev.to/dawidbeno/understanding-yield-in-pytest-fixtures-4m38

THANK YOU!