ti
eng

MENU

Using json-schema for REST API endpoint tests

14 November 2017 on python, django, testing, json-schema, API

Testing REST APIs is a bit like the Wild West. There are tons of ways to do it, different approaches depending on the development language, but no established best practice. This post starts by mentioning a typical way of testing JSON API endpoints that, while works, can be improved. Then, we'll modify the code to apply a better approach taking advantage of an output format specification.

Let's assume the following scenario. We start with a very simple entity with some properties:

```
class MyEntity:

    @property
    def id(self) -> str:
        return "my entity id"

    @property
    def a_property(self) -> bool:
        return True

    @property
    def another_property(self) -> int:
        return 13
```

To separate concerns, we have built a `Presenter` class that handles preparations for a format that can be used from our API endpoints (Django views that will output JSON):

```
from typing import Dict

from .my_entity import MyEntity

```

```
class MyEntityPresenter:

    @staticmethod
    def to_dict(entity: MyEntity) -> Dict:
        return {
            "id": entity.id,
            "a_property": entity.a_property,
            "another_property": entity.another_property
        }
```

And we have a Django view that we'll assume it is already routed:

```
from django.http import HttpRequest, JsonResponse

from .my_entity import MyEntity
from .my_entity_presenter import MyEntityPresenter



def entity_data_json_view(request: HttpRequest, *args:
Any, **kwargs: Any) -> JsonResponse:
    """
    This is our main example endpoint.
    For the sake of the test we assume it is routed to
`/entities/`
    """
    my_entity = MyEntity()
    return
JsonResponse(MyEntityPresenter.to_dict(my_entity))
```

JSON-checking tests

Having those components, the following is a common way of testing JSON outputs:

```
import json
from typing import Any


def test_my_entity_valid_json_output(settings: Any, client: Any) -> None:
    expected_output = '{"a_property": true, "id": "my entity id", "another_property": 13}'

    response = client.get("/entities/")
    assert response.status_code == 200

    response_data = json.loads(response.content)
    assert json.loads(expected_output) == response_data
```

The problem with this approach is that the test is very fragile. Any time you change the output format, you need to go and edit a stringified JSON. And even if you setup the expected data as a dictionary that you then dump to JSON to compare, it is still cumbersome and error-prone, as you will be often coming back to update the expectation of the test (which otherwise would break).

JSON-Schema based tests

Let's introduce JSON-schema. For those of you that in the past have worked with XML, JSON-schema is to JSON what XSD is to XML, a documented and explicit way to define the structure and data types that you will find inside JSON documents.

In order to accommodate it, we'll update the Presenter class to contain also the schema definition:

```
from typing import Dict

from .my_entity import MyEntity


class MyEntityPresenter:

    @staticmethod
    def json_schema() -> Dict:
        return {
            "type": "object",
            "properties": {
                "id": {"type": "string"},
                "a_property": {"type": "boolean"},
                "another_property": {"type": "number"}
            },
            "required": ["id", "another_property",
"another_property"]
        }

    @staticmethod
    def to_dict(entity: MyEntity) -> Dict:
        return {
            "id": entity.id,
            "a_property": entity.a_property,
            "another_property": entity.another_property
        }
```

By placing both the code that creates the output dictionary and the json schema on the same file, we make updating both places in case of any change easy to remember. Our recommendation is to always start updating the schema so

the test will fail if you forget to reflect the change at the `to_dict` method.
There are many more options to make the schema definition really strict, from optional values to enumerations or value ranges, and of course it supports nested entities. Check out the JSON-schema documentation for all the information.

We modify the Django views to use the new classes and provide a nice helper endpoint:

```
from django.http import HttpRequest
from django.http import JsonResponse

from .my_entity import MyEntity
from .my_entity_presenter import MyEntityPresenter


def entity_data_json_view(request: HttpRequest, *args:
Any, **kwargs: Any) -> JsonResponse:
    """
    This is our main example endpoint.
    For the sake of the test we assume it is routed to
`/entities/`
    """
    my_entity = MyEntity()
    return
JsonResponse(MyEntityPresenter.to_dict(my_entity))


def entity_data_json_schema_view(request: HttpRequest,
*args: Any, **kwargs: Any) -> JsonResponse:
    """
    This endpoint allows to output the expected json-
```

```
schema.
    It is useful both for debugging purposes and for
when
    you wish to provide a public interface to the
endpoint.
    It can also easily be adapted to be used in tools
like Swagger.
    """
    return
JsonResponse(MyEntityPresenter.json_schema())
```

Now to the proper test. As every presenter will contain the definition of the contract it expects to be fulfilled, we simply need to test that a call to the entity endpoint returns a JSON that at minimum contains the JSON schema fields and structure (it can have more fields).

```
import json
from typing import Any

from jsonschema import validate

from .my_entity_presenter import MyEntityPresenter


def test_my_entity_valid_json_output(settings: Any,
client: Any) -> None:
    response = client.get("/entities/")
    assert response.status_code == 200

    response_data = json.loads(response.content)
    # raises exception upon any validation error
    validate(response_data,
MyEntityPresenter.json_schema())
    assert response_data["id"] == "my entity id"
    assert response_data["a_property"] is True
    assert response_data["another_property"] == 13
```

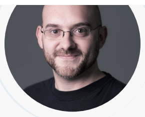
Schema checks are very fast, so nothing forbids you from also using them in other places, like validation of input data, or across microservices boundaries. You could even setup a mechanism to expose JSON-schema contracts from your services, so other services could fetch and run tests against them without actually needing to boot up multiple services (what is usually known as contract-based integration testing).

Additional notes

These are the requirements we've used to run this or a similar example:

```
Django==1.11
uwsgi>=2.0
# Following requirements only for testing
pytest>=3.0
pytest-django>=3.1
jsonschema>=2.6.0
```

You might be wondering why we use `pytest` and `pytest-django`. The answer is that if you take a look at the test method arguments, `settings` and `client`, instead of having to manually load Django settings (for example, to change them for certain tests) or setup a test client to run http requests to your API, `pytest-django` provides both fixtures automatically to every test.

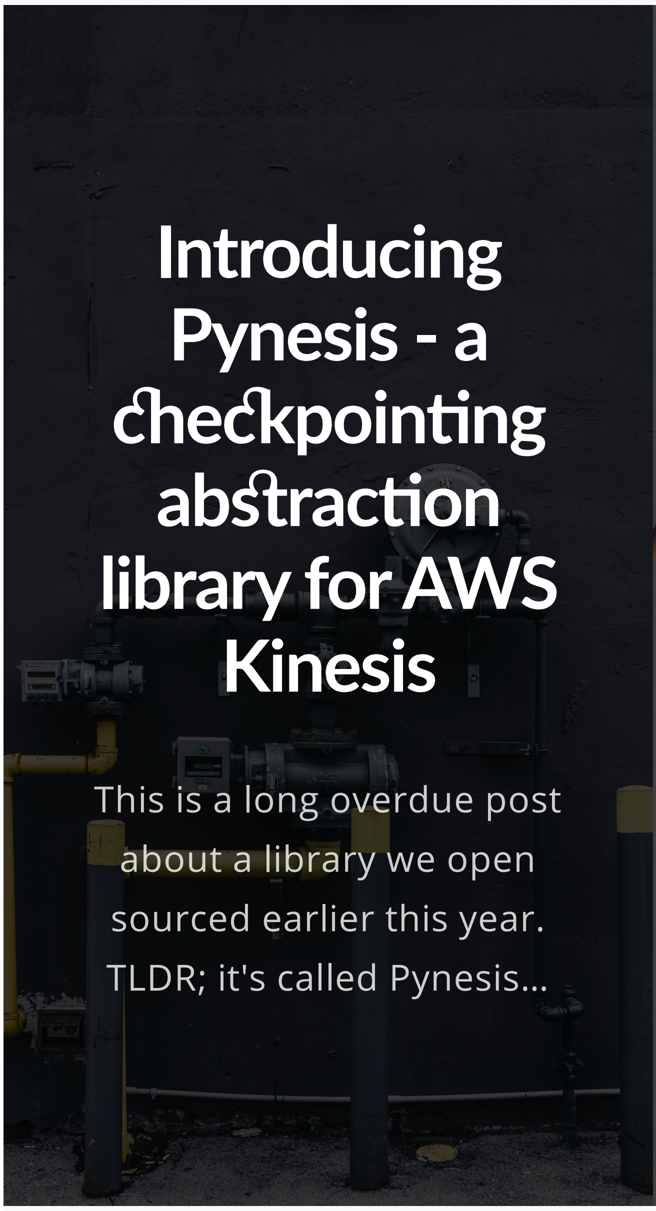


Diego Muñoz 'Kartones'

Read more posts by this author.

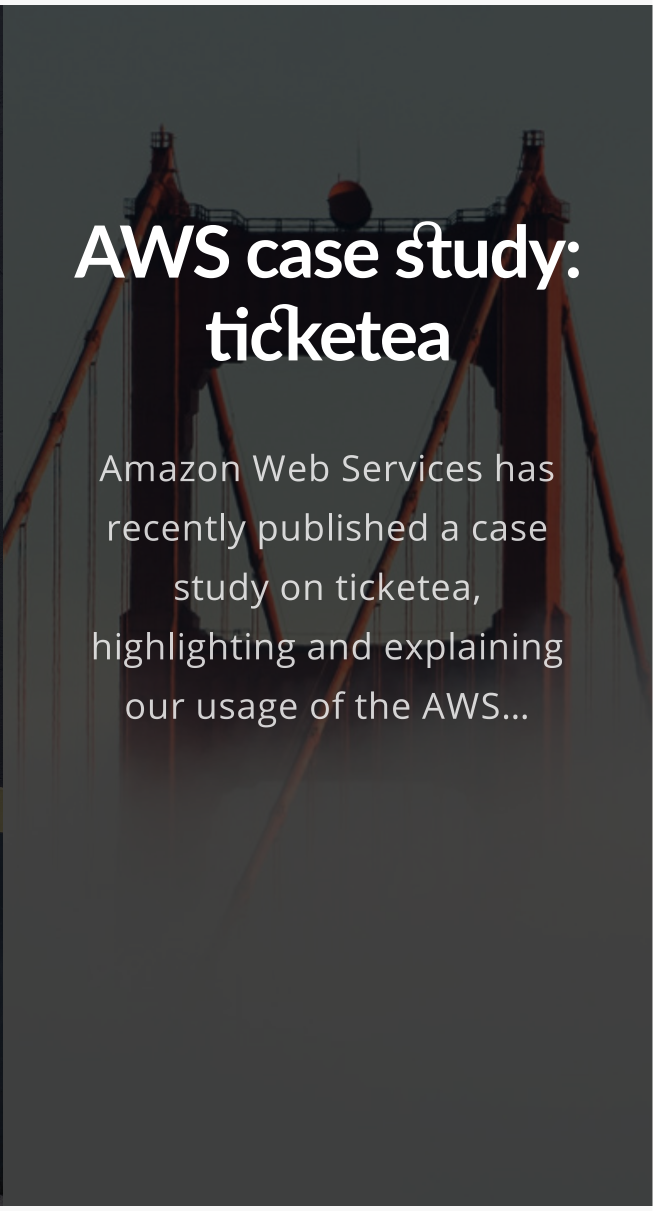
0.0.0.0, Madrid   https://kartones.net

Share this post



Introducing Pynesis - a checkpointing abstraction library for AWS Kinesis

This is a long overdue post about a library we open sourced earlier this year. TLDR; it's called Pynesis...



AWS case study: ticketea

Amazon Web Services has recently published a case study on ticketea, highlighting and explaining our usage of the AWS...