Pyramid Mock Server Documentation

Release v0.0.3

lauris

Feb 12, 2019

Contents

CHAPTER 1

Overview

This package allows to quickly setup a pyramid application from carefully named json mock responses files and a list of endpoints that should exist.

The initial rationale for this package was to serve mocks to clients for endpoints that where not yet developed, but for which the Swagger 2.0 spec was already existing.

As such it as some utility to easily read the list of endpoints from a OpenAPI spec.

`pyramid_mock_server` allow to create mock replica of your services.

CHAPTER 2

Minimal setup

Write a pyramid application as close as possible from you live setup (ports etc.) and add the following

```
config = Configurator(settings={
    # ...

    # pyramid_mock_server config
    'pyramid_mock_server.mock_responses_path': responses_directory, # path to the␣
↪directory where your mocks live
    'pyramid_mock_server.resources': [('/status', 'GET')],  # pairs of endpoint paths,
↪ HTTP verbs that the mocks describe

    # ...
})
config.include('pyramid_mock_server')
```

CHAPTER 3

Setup

3.1 Configuration

```
config = Configurator(settings={
    # ...

    # pyramid_mock_server config
    'pyramid_mock_server.mock_responses_path': responses_directory,
    'pyramid_mock_server.resources': [('/healthcheck', 'GET')],
    'pyramid_mock_server.get_resources_from_pyramid_swagger_2_0_schema': True,
    'pyramid_mock_server.excluded_paths': ['/status', 'swagger.json'],
    'pyramid_mock_server.custom_view_packages': [my_custom_view_packages_1, my_custom_
↪view_packages_2],

    # ...
})
config.include('pyramid_mock_server')
```

- `pyramid_mock_server.mock_responses_path` path to the directory where your mocks live (Required)
- `pyramid_mock_server.resources` pairs of endpoint path, HTTP verbs that the mocks describe (default: `[]`)
- `pyramid_mock_server.get_resources_from_pyramid_swagger_2_0_schema` if `True`, reads the `pyramid_swagger` swagger 2.0 schema to generate resources. (Optional, default: `False`)
- `pyramid_mock_server.excluded_paths` paths that might be in resources (or swagger) but you want ignored. (Optional, default: `None`)
- `pyramid_mock_server.custom_view_packages` array of packages to import custom views from (Optional, default: `None`)

Note: If you enable `pyramid_mock_server.get_resources_from_pyramid_swagger_2_0_schema` configuration make sure that *pyramid_swagger* is installed on your virtual environment.

You could use `pyramid-swagger` extra dependency while installing `pyramid-mock-server` (`pip install pyramid-mock-server[pyramid-swagger]`).

3.2 Custom Views

To mock behavior that rely on non-url parameters, you will have to write custom views. The library helps you to integrate these views within the automatic view generation.

Failure to use these decorators will results in the view auto-generation not working as expected.

`register_custom_view` takes 2 arguments, the `url` and `HTTP verb` this view applies for, in a similar fashion of what you give to `setup_routes_views` initially.

3.2.1 Writing Custom views

```
@register_custom_view('/foo/bar', 'POST')
def custom_view(request):
    return Response(
        '{"message":"custom"}',
        content_type='application/json',
        charset='UTF-8',
    )
```

3.2.2 Discovering Custom Views

Custom views are discovered the same way as regular views are, by looking for the decorator in all function in a package (and its subpackages). To make that discovery possible, remember to setup `'pyramid_mock_server.custom_view_packages'`

CHAPTER 4

Writing Mocks

4.1 Discovery

All the json files in the `responses_directory` directory and its subdirectory will be inspected. All the files that fit the file naming convention will be matched against the given swagger resources.

The subdirectories structure is ignored, so you can organize your mock files as you see fit.

4.2 File naming convention

Generic structure `<transformed_url>(_query<query_args>)_response(.<http_response_code>).<http_verb>.json`

To be able to match urls and mocks following rules apply:

- A mock file name pattern is basically the url from the swagger spec, with / being replaced by _.
- You must also specify the HTTP verb that this response correspond to.
- Arguments are specified the same way as the swagger doc {business_id}.
- Optionaly specific query arguments can be defined for a response.
- Mock ends with _response (this is practical for templating).

eg. A mock for `/business/{business_id}/detail/v1` as `GET` would be named `business_{business_id}_detail_v1_response.GET.json`

You can specify different variations depending on the path arguments by giving the argument value(s) this mock should be returned for

eg. `business_{business_id#foo}_detail_v1_response.GET.json` would be the returned response when calling `/business/foo/detail/v1` on the mock server.

You can also specify a response code for this response (nothing is interpreted as 200):

eg. `business_{business_id#404}_detail_v1_response.404.GET.json` would be the returned 404 response when calling `/business/404/detail/v1` on the mock server.

You can also specify some query arguments that need to be present for a response to be returned: eg. `business_v2_query{business_ids#32,33,34}{with_info#1}_response.GET.json` would be the returned when calling `/business/v2?business_ids=32,33,34&with_info=1` on the mock server.

4.3 Templating

All mocks can use the jinja2 templating language. This allow to include mocks from one within another, or to have templates inheritance. Jinja will consider all json files in the `responses_path` as potential templates, and you can reference any file in there from any file by path from the `responses_path` directory.

4.3.1 Extra Operations

We define 2 extra operations for json objects. These expects a dictionary as top object of the json.

`base.json`

```
{
  "value": 0,
  "attr1": {
    "value": 1,
    "attr2": {
      "value": 2
    }
  }
}
```

Override

This is similar to the `update` for a python dictionary. The key from the original object will be overwritten by the key from the override dictionary.

```
{% override "base.json" %}
{
    "extra": "extra",
    "attr1": "override"
}
{% endoverride %}
```

Render as:

```
{
  "value": 0,
  "attr1": "override"
}
```

Patch

This is more of a "soft" update. All the keys in the `patch` are inserted if they are not present, and if the key is present in the original dictionary, we try to recursively merge contents, preserving as much of the original values as possible.

```
{% patch "base.json" %}
{
```

(continues on next page)

(continued from previous page)

```
  "extra": "extra",
  "value": "patch",
  "attr1": {
    "attr2": {
      "value": "patch"
    }
  }
}
{% endpatch %}
```

Render as:

```
{
  "extra": "extra",
  "value": "patch",
  "attr1": {
    "value": 1,
    "attr2": {
      "value": "patch"
    }
  }
}
```

CHAPTER 5

Using Mock Server in test

These are two tests sample that you might want to run to be sure your service is correct. The first one ensures that all of your endpoints have at least one mock. The second one calls all the mocks that you have defined with the correct parameters.

> For the latest, we recommend your service integrates `pyramid_swagger`, and that you set it up for your mock server as well. You that will thus guaranty that all the mocks you return are compliant to your swagger spec.

```
import pytest
from pyramid_mock_server.mock_loader import load_responses
from pyramid_mock_server.response_collection import ResponseCollection
from pyramid_mock_server.swagger_util import get_all_mocks_operations

# Let's turn off request validation for testing, so we can automagically
# generate tests for every added simple route, with the same config
# for pyramid swagger as prod
@pytest.yield_fixture(scope='session', autouse=True)
def mock_request_validation():
    with mock.patch('pyramid_swagger.tween.swaggerize_request'):
        yield


@pytest.mark.parametrize(
    'path, request_method',
    swagger_resources,
)
def test_all_views(mock_app, path, request_method):
    """
    Test that all views (URL, HTTP operation) respond HTTP 200.
    """
    mock_app.request(path, method=request_method, status=200)


@pytest.mark.parametrize(
```

(continues on next page)

(continued from previous page)

```
    'path, request_method',
    get_all_mocks_operations(
        ResponseCollection(load_responses(responses_path)),
        swagger_resources,
    ),
)
def test_all_mocks(mock_app, path, request_method):
    """
    Test that all views (URL, HTTP operation) respond HTTP 200.
    """
    mock_app.request(path, method=request_method, status=200)
```

CHAPTER 6

Enhance Swagger Specs

While working on the implementation of Swagger 2.0 endpoint could nice to have examples of responses in the swagger specs themselves

```
swagger: "2.0"
info:
    title: Swagger Mock Server Test Spec
    version: 1.0.0
produces: [application/json]
paths:
  /endpoint:
    get:
      responses:
        '200':
          description: OK response
          schema:
            type: string
          examples:
            application/json:
              "example of response"
```

`pyramid-mock-server` provides a cli tool, `pyramid-mock-server-spec-enhancer` , to injects mock server responses into the examples section of the swagger specs.

Note: To use the tool you need to install the library with `cli` extra dependency (`pip install pyramid-mock-server[cli]`)