Clean Architectures in Python

Chapter 4 - Add a web application

For your information, Hairdo, a major network is interested in me.

Groundhog Day, 1993

In this chapter, I will go through the creation of an HTTP endpoint for the room list use case. An HTTP endpoint is a URL exposed by a Web server that runs a specific logic and returns values in a standard format.

I will follow the REST recommendation, so the endpoint will return a JSON payload. REST is however not part of the clean architecture, which means that you can choose to model your URLs and the format of returned data according to whatever scheme you prefer.

To expose the HTTP endpoint we need a web server written in Python, and in this case, I chose Flask. Flask is a lightweight web server with a modular structure that provides just the parts that the user needs. In particular, we will not use any database/ORM, since we already implemented our own repository layer.

Flask setup ¶

Let us start updating the requirements files. The file `requirements/prod.txt` shall mention Flask, as this package contains a script that runs a local web-

server that we can use to expose the endpoint

`requirements/prod.txt`

```
Flask
```

The file `requirements/test.txt` will contain the pytest extension to work with Flask (more on this later)

`requirements/test.txt`

```
-r prod.txt
pytest
tox
coverage
pytest-cov
pytest-flask
```

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c04-s01

Remember to run `pip install -r requirements/dev.txt` again after those changes to install the new packages in your virtual environment.

The setup of a Flask application is not complex, but there are a lot of concepts involved, and since this is not a tutorial on Flask I will run quickly through these steps. I will provide links to the Flask documentation for every concept, though. If you want to dig a bit deeper in this matter you can read my series of posts Flask Project Setup: TDD, Docker, Postgres and more.

The Flask application can be configured using a plain Python object (documentation), so I created the file `application/config.py` that contains this code

application/config.py

```
import os

basedir = os.path.abspath(os.path.dirname(__file__))


class Config(object):
    """Base configuration"""


class ProductionConfig(Config):
    """Production configuration"""


class DevelopmentConfig(Config):
    """Development configuration"""


class TestingConfig(Config):
    """Testing configuration"""

    TESTING = True
```

Read this page to know more about Flask configuration parameters.

Now we need a function that initialises the Flask application (documentation), configures it, and registers the blueprints (documentation). The file `application/app.py` contains the following code, which is an app factory

application/app.py

```
from flask import Flask
```

```
from application.rest import room


def create_app(config_name):

    app = Flask(__name__)

    config_module = f"application.config.{config_name.capita

    app.config.from_object(config_module)

    app.register_blueprint(room.blueprint)

    return app
```

Source code

https://github.com/pycabook/rentomatic/tree/ed2-c04-s02

Test and create an HTTP endpoint ¶

Before we create the proper setup of the webserver, we want to create the endpoint that will be exposed. Endpoints are ultimately functions that are run when a user sends a request to a certain URL, so we can still work with TDD, as the final goal is to have code that produces certain results.

The problem we have testing an endpoint is that we need the webserver to be up and running when we hit the test URLs. The webserver itself is an external system so we won't test it, but the code that provides the endpoint is part of our application[1]. It is actually a gateway, that is an interface that allows an HTTP framework to access the use cases.

The extension `pytest-flask` allows us to run Flask, simulate HTTP requests, and test the HTTP responses. This extension hides a lot of automation, so it might be considered a bit "magic" at a first glance. When you install it some fixtures like `client` are available automatically, so you don't need to import them. Moreover, it tries to access another fixture named `app` that you have to define. This is thus the first thing to do.

Fixtures can be defined directly in your test file, but if we want a fixture to be globally available the best place to define it is the file `conftest.py` which is automatically loaded by pytest. As you can see there is a great deal of automation, and if you are not aware of it you might be surprised by the results, or frustrated by the errors.

`tests/conftest.py`

```
import pytest


from application.app import create_app


@pytest.fixture
def app():
    app = create_app("testing")

    return app
```

The function `app` runs the app factory to create a Flask app, using the configuration `testing`, which sets the flag `TESTING` to `True`. You can find the description of these flags in the official documentation.

At this point, we can write the test for our endpoint.

`tests/rest/test_room.py`

```
import json
from unittest import mock

from rentomatic.domain.room import Room

room_dict = {
    "code": "3251a5bd-86be-428d-8ae9-6e51a8048c33",
    "size": 200,
    "price": 10,
    "longitude": -0.09998975,
    "latitude": 51.75436293,
}

rooms = [Room.from_dict(room_dict)]


@mock.patch("application.rest.room.room_list_use_case")
def test_get(mock_use_case, client):
    mock_use_case.return_value = rooms

    http_response = client.get("/rooms")

    assert json.loads(http_response.data.decode("UTF-8")) ==
    mock_use_case.assert_called()
    assert http_response.status_code == 200
    assert http_response.mimetype == "application/json"
```

Let's comment it section by section.

```
tests/rest/test_room.py
```

```
import json
from unittest import mock

from rentomatic.domain.room import Room

room_dict = {
    "code": "3251a5bd-86be-428d-8ae9-6e51a8048c33",
    "size": 200,
    "price": 10,
    "longitude": -0.09998975,
    "latitude": 51.75436293,
}

rooms = [Room.from_dict(room_dict)]
```

The first part contains some imports and sets up a room from a dictionary. This way we can later directly compare the content of the initial dictionary with the result of the API endpoint. Remember that the API returns JSON content, and we can easily convert JSON data into simple Python structures, so starting from a dictionary will come in handy.

tests/rest/test_room.py

```
@mock.patch("application.rest.room.room_list_use_case")
def test_get(mock_use_case, client):
```

This is the only test that we have for the time being. During the whole test, we mock the use case, as we are not interested in running it, as it has been already tested elsewhere. We are however interested in checking the arguments passed to the use case, and a mock can provide this information. The test receives the mock from the decorator `patch` and the fixture `client`, which is one of the fixtures provided by `pytest-flask`. The fixture automatically loads `app`, which we defined in `conftest.py`, and is an object that simulates an

HTTP client that can access the API endpoints and store the responses of the server.

tests/rest/test_room.py

```
    mock_use_case.return_value = rooms

    http_response = client.get("/rooms")

    assert json.loads(http_response.data.decode("UTF-8")) ==
    mock_use_case.assert_called()
    assert http_response.status_code == 200
    assert http_response.mimetype == "application/json"
```

The first line initialises the mock use case, instructing it to return the fixed `rooms` variable that we created previously. The central part of the test is the line where we `get` the API endpoint, which sends an HTTP GET request and collects the server's response.

After this, we check that the data contained in the response is a JSON that contains the data in the structure `room_dict`, that the method use_case has been called, that the HTTP response status code is 200, and last that the server sends the correct MIME type back.

It's time to write the endpoint, where we will finally see all the pieces of the architecture working together, as they did in the little CLI program that we wrote previously. Let me show you a template for the minimal Flask endpoint we can create

```
blueprint = Blueprint('room', __name__)


@blueprint.route('/rooms', methods=['GET'])
def room_list():
```

```
    [LOGIC]
    return Response([JSON DATA],
                    mimetype='application/json',
                    status=[STATUS])
```

As you can see the structure is really simple. Apart from setting the blueprint, which is the way Flask registers endpoints, we create a simple function that runs the endpoint, and we decorate it assigning the enpoint `/rooms` that serves `GET` requests. The function will run some logic and eventually return a `Response` that contains JSON data, the correct MIME type, and an HTTP status that represents the success or failure of the logic.

The above template becomes the following code

application/rest/room.py

```
import json

from flask import Blueprint, Response

from rentomatic.repository.memrepo import MemRepo
from rentomatic.use_cases.room_list import room_list_use_cas
from rentomatic.serializers.room import RoomJsonEncoder

blueprint = Blueprint("room", __name__)

rooms = [
    {
        "code": "f853578c-fc0f-4e65-81b8-566c5dffa35a",
        "size": 215,
        "price": 39,
        "longitude": -0.09998975,
        "latitude": 51.75436293,
    },
```

```
    {
        "code": "fe2c3195-aeff-487a-a08f-e0bdc0ec6e9a",
        "size": 405,
        "price": 66,
        "longitude": 0.18228006,
        "latitude": 51.74640997,
    },
    {
        "code": "913694c6-435a-4366-ba0d-da5334a611b2",
        "size": 56,
        "price": 60,
        "longitude": 0.27891577,
        "latitude": 51.45994069,
    },
    {
        "code": "eed76e77-55c1-41ce-985d-ca49bf6c0585",
        "size": 93,
        "price": 48,
        "longitude": 0.33894476,
        "latitude": 51.39916678,
    },
]


@blueprint.route("/rooms", methods=["GET"])
def room_list():
    repo = MemRepo(rooms)
    result = room_list_use_case(repo)

    return Response(
        json.dumps(result, cls=RoomJsonEncoder),
        mimetype="application/json",
```

```
        status=200,
    )
```

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c04-s03

Please note that I initialised the memory storage with the same list used for the script `cli.py`. Again, the need of initialising the storage with data (even with an empty list) is due to the limitations of the storage `MemRepo`. The code that runs the use case is

application/rest/room.py

```
def room_list():
    repo = MemRepo(rooms)
    result = room_list_use_case(repo)
```

which is exactly the same code that we used in the command-line interface. The last part of the code creates a proper HTTP response, serializing the result of the use case using `RoomJsonEncoder`, and setting the HTTP status to 200 (success)

application/rest/room.py

```
    return Response(
        json.dumps(result, cls=RoomJsonEncoder),
        mimetype="application/json",
        status=200,
    )
```

This shows you the power of the clean architecture in a nutshell. Writing a CLI interface or a Web service is different only in the presentation layer, not in the logic, which is the same, as it is contained in the use case.

Now that we defined the endpoint, we can finalise the configuration of the webserver, so that we can access the endpoint with a browser. This is not strictly part of the clean architecture, but as I did with the CLI interface I want you to see the final result, to get the whole picture and also to enjoy the effort you put in following the whole discussion up to this point.

WSGI ¶

Python web applications expose a common interface called Web Server Gateway Interface or WSGI. So to run the Flask development web server, we have to define a `wsgi.py` file in the main folder of the project, i.e. in the same directory of the file `cli.py`

wsgi.py

```
import os

from application.app import create_app

app = create_app(os.environ["FLASK_CONFIG"])
```

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c04-s04

When you run the Flask Command Line Interface (documentation), it automatically looks for a file named `wsgi.py` and loads it, expecting it to contain a vari-

able named `app` that is an instance of the object `Flask`. As the function `create_app` is a factory we just need to execute it.

At this point, you can execute `FLASK_CONFIG="development" flask run` in the directory that contains this file and you should see a nice message like

```
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

At this point, you can point your browser to http://127.0.0.1:5000/rooms and enjoy the JSON data returned by the first endpoint of your web application.

I hope you can now appreciate the power of the layered architecture that we created. We definitely wrote a lot of code to "just" print out a list of models, but the code we wrote is a skeleton that can easily be extended and modified. It is also fully tested, which is a part of the implementation that many software projects struggle with.

The use case I presented is purposely very simple. It doesn't require any input and it cannot return error conditions, so the code we wrote completely ignored input validation and error management. These topics are however extremely important, so we need to discuss how a clean architecture can deal with them.

1 We could, in theory, create a pure component that receives parameters and returns a JSON object, and then wrap this component into an endpoint. This way, the component would be strictly part of the internal system and the endpoint of the external one, but both would have to be created in the Gateway layer. This looks overkill, at least for the simple example we are discussing here, so I will keep them together and test them as a single component.

Last update: 20/08/2021

Section 6 of Clean Architectures in Python

Previous

- Introduction
- About the book
- Chapter 1 - A day in the life of a clean system
- Chapter 2 - Components of a clean architecture
- Chapter 3 - A basic example

Next

- Chapter 5 - Error management
- Chapter 6 - Integration with a real external system - Postgres
- Chapter 7 - Integration with a real external system - MongoDB
- Chapter 8 - Run a production-ready system
- Changelog
- Colophon