Clean Architectures in Python

Chapter 5 - Error management

You sent them out there and you didn't even warn them! Why didn't you warn them, Burke?

Aliens, 1986

In every software project, a great part of the code is dedicated to error management, and this code has to be rock solid. Error management is a complex topic, and there is always a corner case that we left out, or a condition that we supposed could never fail, while it does.

In a clean architecture, the main process is the creation of use cases and their execution. This is, therefore, the main source of errors, and the use cases layer is where we have to implement the error management. Errors can obviously come from the domain models layer, but since those models are created by the use cases the errors that are not managed by the models themselves automatically become errors of the use cases.

Request and responses ¶

We can divide the error management code into two different areas. The first one represents and manages **requests**, that is, the input data that reaches our use case. The second one covers the way we return results from the use case through **responses**, the output data. These two concepts shouldn't be confused with HTTP requests and responses, even though there are similarities. We are now considering the way data can be passed to and received from use cases, and how to manage errors. This has nothing to do with the possible use of this architecture to expose an HTTP API.

Request and response objects are an important part of a clean architecture, as they transport call parameters, inputs and results from outside the application into the use cases layer.

More specifically, requests are objects created from incoming API calls, thus they shall deal with things like incorrect values, missing parameters, wrong formats, and so on. Responses, on the other hand, have to contain the actual results of the API calls, but shall also be able to represent error cases and deliver rich information on what happened.

The actual implementation of request and response objects is completely free, the clean architecture says nothing about them. The decision on how to pack and represent data is up to us.

To start working on possible errors and understand how to manage them, I will expand `room_list_use_case` to support filters that can be used to select a subset of the `Room` objects in storage.

The filters could be, for example, represented by a dictionary that contains attributes of the model `Room` and the logic to apply to them. Once we accept such a rich structure, we open our use case to all sorts of errors: attributes that do not exist in the model, thresholds of the wrong type, filters that make the storage layer crash, and so on. All these considerations have to be taken into account by the use case.

Basic structure ¶

We can implement structured requests before we expand the use case to accept filters. We just need a class `RoomListRequest` that can be initialised without parameters, so let us create the file `tests/requests/test_room_list.py` and put there a test for this object.

tests/requests/test_room_list.py

```python
from rentomatic.requests.room_list import RoomListRequest


def test_build_room_list_request_without_parameters():
    request = RoomListRequest()

    assert bool(request) is True


def test_build_room_list_request_from_empty_dict():
    request = RoomListRequest.from_dict({})

    assert bool(request) is True
```

While at the moment this request object is basically empty, it will come in handy as soon as we start having parameters for the list use case. The code of the class `RoomListRequest` is the following

rentomatic/requests/room_list.py

```python
class RoomListRequest:
    @classmethod
    def from_dict(cls, adict):
        return cls()

    def __bool__(self):
        return True
```

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c05-s01

The response object is also very simple since for the moment we just need to return a successful result. Unlike the request, the response is not linked to any particular use case, so the test file can be named `tests/test_responses.py`

tests/test_responses.py

```python
from rentomatic.responses import ResponseSuccess


def test_response_success_is_true():
    assert bool(ResponseSuccess()) is True
```

and the actual response object is in the file `rentomatic/responses.py`

rentomatic/responses.py

```python
class ResponseSuccess:
    def __init__(self, value=None):
        self.value = value

    def __bool__(self):
        return True
```

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c05-s02

With these two objects, we just laid the foundations for richer management of input and outputs of the use case, especially in the case of error conditions.

Requests and responses in a use case

Let's implement the request and response objects that we developed into the use case. To do this, we need to change the use case so that it accepts a request and return a response. The new version of `tests/use_cases/test_room_list.py` is the following

tests/use_cases/test_room_list.py

```
import pytest
import uuid
from unittest import mock

from rentomatic.domain.room import Room
from rentomatic.use_cases.room_list import room_list_use_cas
from rentomatic.requests.room_list import RoomListRequest


@pytest.fixture
def domain_rooms():
    room_1 = Room(
        code=uuid.uuid4(),
        size=215,
        price=39,
        longitude=-0.09998975,
        latitude=51.75436293,
    )

    room_2 = Room(
        code=uuid.uuid4(),
        size=405,
        price=66,
        longitude=0.18228006,
        latitude=51.74640997,
    )

    room_3 = Room(
        code=uuid.uuid4(),
        size=56,
        price=60,
        longitude=0.27891577,
        latitude=51.45994069,
    )

    room_4 = Room(
        code=uuid.uuid4(),
        size=93,
        price=48,
        longitude=0.33894476,
        latitude=51.39916678,
    )

    return [room_1, room_2, room_3, room_4]


def test_room_list_without_parameters(domain_rooms):
    repo = mock.Mock()
    repo.list.return_value = domain_rooms

    request = RoomListRequest()

    response = room_list_use_case(repo, request)

    assert bool(response) is True
    repo.list.assert_called_with()
    assert response.value == domain_rooms
```

And the changes in the use case are minimal. The new version of the file `rentomatic/use_cases/room_list.py` is the following

rentomatic/use_cases/room_list.py

```
from rentomatic.responses import ResponseSuccess


def room_list_use_case(repo, request):
    rooms = repo.list()
    return ResponseSuccess(rooms)
```

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c05-s03

Now we have a standard way to pack input and output values, and the above pattern is valid for every use case we can create. We are still missing some features, however, because so far requests and responses are not used to perform error management.

Request validation ¶

The parameter `filters` that we want to add to the use case allows the caller to add conditions to narrow the results of the model list operation, using a notation like `<attribute>__<operator>`. For example, specifying `filters=`

{'price__lt': 100} should return all the results with a price lower than 100.

Since the model Room has many attributes, the number of possible filters is very high. For simplicity's sake, I will consider the following cases:

- The attribute code supports only __eq, which finds the room with the specific code if it exists
- The attribute price supports __eq, __lt, and __gt
- All other attributes cannot be used in filters

The core idea here is that requests are customised for use cases, so they can contain the logic that validates the arguments used to instantiate them. The request is valid or invalid before it reaches the use case, so it is not the responsibility of the latter to check that the input values have proper values or a proper format.

This also means that building a request might result in two different objects, a valid one or an invalid one. For this reason, I decided to split the existing class RoomListRequest into RoomListValidRequest and RoomListInvalidRequest, creating a factory function that returns the proper object.

The first thing to do is to change the existing tests to use the factory.

tests/requests/test_room_list.py

```
from rentomatic.requests.room_list import build_room_list_re


def test_build_room_list_request_without_parameters():
    request = build_room_list_request()

    assert request.filters is None
    assert bool(request) is True
```

```
def test_build_room_list_request_with_empty_filters():
    request = build_room_list_request({})

    assert request.filters == {}
    assert bool(request) is True
```

Next, I will test that passing the wrong type of object as `filters` or that using incorrect keys results in an invalid request

```
tests/requests/test_room_list.py
```

```
def test_build_room_list_request_with_invalid_filters_parame
    request = build_room_list_request(filters=5)

    assert request.has_errors()
    assert request.errors[0]["parameter"] == "filters"
    assert bool(request) is False


def test_build_room_list_request_with_incorrect_filter_keys(
    request = build_room_list_request(filters={"a": 1})

    assert request.has_errors()
    assert request.errors[0]["parameter"] == "filters"
    assert bool(request) is False
```

Last, I will test the supported and unsupported keys

```
tests/requests/test_room_list.py
```

```
import pytest

...

@pytest.mark.parametrize(
    "key", ["code__eq", "price__eq", "price__lt", "price__gt
)
def test_build_room_list_request_accepted_filters(key):
    filters = {key: 1}

    request = build_room_list_request(filters=filters)

    assert request.filters == filters
    assert bool(request) is True


@pytest.mark.parametrize("key", ["code__lt", "code__gt"])
def test_build_room_list_request_rejected_filters(key):
    filters = {key: 1}

    request = build_room_list_request(filters=filters)

    assert request.has_errors()
    assert request.errors[0]["parameter"] == "filters"
    assert bool(request) is False
```

Note that I used the decorator `pytest.mark.parametrize` to run the same test on multiple values.

Following the TDD approach, adding those tests one by one and writing the code that passes them, I come up with the following code

```
rentomatic/requests/room_list.py
```

```python
from collections.abc import Mapping


class RoomListInvalidRequest:
    def __init__(self):
        self.errors = []

    def add_error(self, parameter, message):
        self.errors.append({"parameter": parameter, "message

    def has_errors(self):
        return len(self.errors) > 0

    def __bool__(self):
        return False


class RoomListValidRequest:
    def __init__(self, filters=None):
        self.filters = filters

    def __bool__(self):
        return True


def build_room_list_request(filters=None):
    accepted_filters = ["code__eq", "price__eq", "price__lt"
    invalid_req = RoomListInvalidRequest()

    if filters is not None:
        if not isinstance(filters, Mapping):
            invalid_req.add_error("filters", "Is not iterabl
            return invalid_req
        for key, value in filters.items():
            if key not in accepted_filters:
                invalid_req.add_error(
                    "filters", "Key {} cannot be used".forma
                )

        if invalid_req.has_errors():
            return invalid_req

    return RoomListValidRequest(filters=filters)
```

The introduction of the factory makes one use case test fails. The new version of that test is

```
tests/use_cases/test_room_list.py
```

```
...

from rentomatic.requests.room_list import build_room_list_re

...

def test_room_list_without_parameters(domain_rooms):
    repo = mock.Mock()
    repo.list.return_value = domain_rooms

    request = build_room_list_request()

    response = room_list_use_case(repo, request)

    assert bool(response) is True
    repo.list.assert_called_with()
    assert response.value == domain_rooms
```

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c05-s04

Responses and failures ¶

There is a wide range of errors that can happen while the use case code is executed. Validation errors, as we just discussed in the previous section, but also business logic errors or errors that come from the repository layer or other external systems that the use case interfaces with. Whatever the error, the use case shall always return an object with a known structure (the response), so we need a new object that provides good support for different types of failures.

As happened for the requests there is no unique way to provide such an object, and the following code is just one of the possible solutions. First of all, after some necessary imports, I test that responses have a boolean value

tests/test_responses.py

```
from rentomatic.responses import (
    ResponseSuccess,
    ResponseFailure,
    ResponseTypes,
    build_response_from_invalid_request,
)
from rentomatic.requests.room_list import RoomListInvalidReq

SUCCESS_VALUE = {"key": ["value1", "value2"]}
```

```
GENERIC_RESPONSE_TYPE = "Response"
GENERIC_RESPONSE_MESSAGE = "This is a response"


def test_response_success_is_true():
    response = ResponseSuccess(SUCCESS_VALUE)

    assert bool(response) is True


def test_response_failure_is_false():
    response = ResponseFailure(
        GENERIC_RESPONSE_TYPE, GENERIC_RESPONSE_MESSAGE
    )

    assert bool(response) is False
```

Then I test the structure of responses, checking `type` and `value`. `ResponseFailure` objects should also have an attribute `message`

tests/test_responses.py

```
def test_response_success_has_type_and_value():
    response = ResponseSuccess(SUCCESS_VALUE)

    assert response.type == ResponseTypes.SUCCESS
    assert response.value == SUCCESS_VALUE


def test_response_failure_has_type_and_message():
    response = ResponseFailure(
        GENERIC_RESPONSE_TYPE, GENERIC_RESPONSE_MESSAGE
    )
    assert response.type == GENERIC_RESPONSE_TYPE
    assert response.message == GENERIC_RESPONSE_MESSAGE
    assert response.value == {
        "type": GENERIC_RESPONSE_TYPE,
        "message": GENERIC_RESPONSE_MESSAGE,
    }
```

The remaining tests are all about `ResponseFailure`. First, a test to check that it can be initialised with an exception

tests/test_responses.py

```
def test_response_failure_initialisation_with_exception():
    response = ResponseFailure(
        GENERIC_RESPONSE_TYPE, Exception("Just an error mess
    )

    assert bool(response) is False
    assert response.type == GENERIC_RESPONSE_TYPE
    assert response.message == "Exception: Just an error mes
```

Since we want to be able to build a response directly from an invalid request, getting all the errors contained in the latter, we need to test that case

tests/test_responses.py

```
def test_response_failure_from_empty_invalid_request():
    response = build_response_from_invalid_request(
        RoomListInvalidRequest()
    )

    assert bool(response) is False
    assert response.type == ResponseTypes.PARAMETERS_ERROR


def test_response_failure_from_invalid_request_with_errors()
    request = RoomListInvalidRequest()
    request.add_error("path", "Is mandatory")
    request.add_error("path", "can't be blank")

    response = build_response_from_invalid_request(request)

    assert bool(response) is False
    assert response.type == ResponseTypes.PARAMETERS_ERROR
    assert response.message == "path: Is mandatory\npath: ca
```

Let's write the classes that make the tests pass

```
rentomatic/responses.py
```

```
class ResponseTypes:
    PARAMETERS_ERROR = "ParametersError"
    RESOURCE_ERROR = "ResourceError"
    SYSTEM_ERROR = "SystemError"
    SUCCESS = "Success"


class ResponseFailure:
    def __init__(self, type_, message):
        self.type = type_
        self.message = self._format_message(message)

    def _format_message(self, msg):
        if isinstance(msg, Exception):
            return "{}: {}".format(
                msg.__class__.__name__, "{}".format(msg)
            )
        return msg

    @property
    def value(self):
        return {"type": self.type, "message": self.message}

    def __bool__(self):
        return False


class ResponseSuccess:
    def __init__(self, value=None):
        self.type = ResponseTypes.SUCCESS
        self.value = value

    def __bool__(self):
        return True


def build_response_from_invalid_request(invalid_request):
    message = "\n".join(
        [
            "{}: {}".format(err["parameter"], err["message"]
            for err in invalid_request.errors
        ]
    )
    return ResponseFailure(ResponseTypes.PARAMETERS_ERROR, m
```

Through the method `_format_message()` we enable the class to accept both string messages and Python exceptions, which is very handy when dealing with external libraries that can raise exceptions we do not know or do not want to manage.

The error types contained in the class `ResponseTypes` are very similar to HTTP errors, and this will be useful later when we will return responses from the web framework. `PARAMETERS_ERROR` signals that something was wrong in the input parameters passed by the request. `RESOURCE_ERROR` signals that the process ended correctly, but the requested resource is not available, for example when reading a specific value from a data storage. Last, `SYSTEM_ERROR` signals that something went wrong with the process itself, and will be used mostly to signal an exception in the Python code.

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c05-s05

Error management in a use case ¶

Our implementation of requests and responses is finally complete, so we can now implement the last version of our use case. The function `room_list_use_case` is still missing a proper validation of the incoming request, and is not returning a suitable response in case something went wrong.

The test `test_room_list_without_parameters` must match the new API, so I added `filters=None` to `assert_called_with`

tests/use_cases/test_room_list.py

```python
def test_room_list_without_parameters(domain_rooms):
    repo = mock.Mock()
    repo.list.return_value = domain_rooms

    request = build_room_list_request()
    response = room_list_use_case(repo, request)

    assert bool(response) is True
    repo.list.assert_called_with(filters=None)
    assert response.value == domain_rooms
```

There are three new tests that we can add to check the behaviour of the use case when `filters` is not `None`. The first one checks that the value of the key `filters` in the dictionary used to create the request is actually used when calling the repository. These last two tests check the behaviour of the use case when the repository raises an exception or when the request is badly formatted.

tests/use_cases/test_room_list.py

```
import pytest
import uuid
from unittest import mock

from rentomatic.domain.room import Room
from rentomatic.use_cases.room_list import room_list_use_cas
from rentomatic.requests.room_list import build_room_list_re
from rentomatic.responses import ResponseTypes

...

def test_room_list_with_filters(domain_rooms):
    repo = mock.Mock()
    repo.list.return_value = domain_rooms

    qry_filters = {"code__eq": 5}
    request = build_room_list_request(filters=qry_filters)
```

```python
    response = room_list_use_case(repo, request)

    assert bool(response) is True
    repo.list.assert_called_with(filters=qry_filters)
    assert response.value == domain_rooms


def test_room_list_handles_generic_error():
    repo = mock.Mock()
    repo.list.side_effect = Exception("Just an error message

    request = build_room_list_request(filters={})

    response = room_list_use_case(repo, request)

    assert bool(response) is False
    assert response.value == {
        "type": ResponseTypes.SYSTEM_ERROR,
        "message": "Exception: Just an error message",
    }


def test_room_list_handles_bad_request():
    repo = mock.Mock()

    request = build_room_list_request(filters=5)

    response = room_list_use_case(repo, request)

    assert bool(response) is False
    assert response.value == {
        "type": ResponseTypes.PARAMETERS_ERROR,
        "message": "filters: Is not iterable",
    }
```

Now change the use case to contain the new use case implementation that makes all the tests pass

```
rentomatic/use_cases/room_list.py
```

```
from rentomatic.responses import (
    ResponseSuccess,
    ResponseFailure,
    ResponseTypes,
    build_response_from_invalid_request,
)


def room_list_use_case(repo, request):
    if not request:
        return build_response_from_invalid_request(request)
    try:
        rooms = repo.list(filters=request.filters)
        return ResponseSuccess(rooms)
    except Exception as exc:
        return ResponseFailure(ResponseTypes.SYSTEM_ERROR, e
```

As you can see, the first thing that the use case does is to check if the request is valid. Otherwise, it returns a `ResponseFailure` built with the same request object. Then the actual business logic is implemented, calling the repository and returning a successful response. If something goes wrong in this phase the exception is caught and returned as an aptly formatted `ResponseFailure`.

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c05-s06

Integrating external systems ¶

I want to point out a big problem represented by mocks.

As we are testing objects using mocks for external systems, like the repository, no tests fail at the moment, but trying to run the Flask development server will certainly return an error. As a matter of fact, neither the repository nor the HTTP server are in sync with the new API, but this cannot be shown by unit tests if they are properly written. This is the reason why we need integration tests, since external systems that rely on a certain version of the API are running only at that point, and this can raise issues that were masked by mocks.

For this simple project, my integration test is represented by the Flask development server, which at this point crashes. If you run `FLASK_CONFIG="development" flask run` and open http://127.0.0.1:5000/rooms with your browser you will get and Internal Server Error, and on the command line this exception

```
TypeError: room_list_use_case() missing 1 required positiona
```

The same error is returned by the CLI interface. After the introduction of requests and responses we didn't change the REST endpoint, which is one of the connections between the external world and the use case. Given that the API of the use case changed, we need to change the code of the endpoints that call the use case.

The HTTP server

As we can see from the exception above the use case is called with the wrong parameters in the REST endpoint. The new version of the test is

tests/rest/test_room.py

```
import json
from unittest import mock

import pytest

from rentomatic.domain.room import Room
from rentomatic.responses import (
    ResponseFailure,
    ResponseSuccess,
    ResponseTypes,
)

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 = ResponseSuccess(rooms)

    http_response = client.get("/rooms")
    assert json.loads(http_response.data.decode("UTF-8")) ==

    mock_use_case.assert_called()
    args, kwargs = mock_use_case.call_args
    assert args[1].filters == {}

    assert http_response.status_code == 200
    assert http_response.mimetype == "application/json"


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

    http_response = client.get(
        "/rooms?filter_price__gt=2&filter_price__lt=6"
    )

    assert json.loads(http_response.data.decode("UTF-8")) ==

    mock_use_case.assert_called()
    args, kwargs = mock_use_case.call_args
    assert args[1].filters == {"price__gt": "2", "price__lt"

    assert http_response.status_code == 200
    assert http_response.mimetype == "application/json"


@pytest.mark.parametrize(
    "response_type, expected_status_code",
    [
        (ResponseTypes.PARAMETERS_ERROR, 400),
        (ResponseTypes.RESOURCE_ERROR, 404),
        (ResponseTypes.SYSTEM_ERROR, 500),
    ],
)
@mock.patch("application.rest.room.room_list_use_case")
def test_get_response_failures(
    mock_use_case,
    client,
    response_type,
    expected_status_code,
):
    mock_use_case.return_value = ResponseFailure(
        response_type,
        message="Just an error message",
    )

    http_response = client.get("/rooms?dummy_request_string"

    mock_use_case.assert_called()

    assert http_response.status_code == expected_status_code
```

The function `test_get` was already present but has been changed to reflect the use of requests and responses. The first change is that the use case in the mock has to return a proper response

```
mock_use_case.return_value = ResponseSuccess(rooms)
```

and the second is the assertion on the call of the use case. It should be called with a properly formatted request, but since we can't compare requests, we need a way to look into the call arguments. This can be done with

```
mock_use_case.assert_called()
args, kwargs = mock_use_case.call_args
```

```
assert args[1].filters == {}
```

as the use case should receive a request with empty filters as an argument.

The function `test_get_with_filters` performs the same operation but passing a query string to the URL `/rooms`, which requires a different assertion

```
assert args[1].filters == {'price__gt': '2', 'price__lt': '6
```

Both the tests pass with a new version of the endpoint `room_list`

application/rest/room.py

```
import json

from flask import Blueprint, request, Response

from rentomatic.repository.memrepo import MemRepo
from rentomatic.use_cases.room_list import room_list_use_cas
from rentomatic.serializers.room import RoomJsonEncoder
from rentomatic.requests.room_list import build_room_list_re
from rentomatic.responses import ResponseTypes

blueprint = Blueprint("room", __name__)

STATUS_CODES = {
    ResponseTypes.SUCCESS: 200,
    ResponseTypes.RESOURCE_ERROR: 404,
    ResponseTypes.PARAMETERS_ERROR: 400,
    ResponseTypes.SYSTEM_ERROR: 500,
}

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():
    qrystr_params = {
        "filters": {},
    }

    for arg, values in request.args.items():
        if arg.startswith("filter_"):
            qrystr_params["filters"][arg.replace("filter_",

    request_object = build_room_list_request(
        filters=qrystr_params["filters"]
    )

    repo = MemRepo(rooms)
    response = room_list_use_case(repo, request_object)

    return Response(
        json.dumps(response.value, cls=RoomJsonEncoder),
        mimetype="application/json",
        status=STATUS_CODES[response.type],
    )
```

Please note that I'm using a variable named `request_object` here to avoid clashing with the fixture `request` provided by `pytest-flask`. While `request` contains the HTTP request sent to the web framework by the browser, `request_object` is the request we send to the use case.

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c05-s07

The repository

If we run the Flask development webserver now and try to access the endpoint `/rooms`, we will get a nice response that says

```
{"type": "SystemError", "message": "TypeError: list() got an
```

and if you look at the HTTP response[1] you can see an HTTP 500 error, which is exactly the mapping of our `SystemError` use case error, which in turn signals a Python exception, which is in the `message` part of the error.

This error comes from the repository, which has not been migrated to the new API. We need then to change the method `list` of the class `MemRepo` to accept the parameter `filters` and to act accordingly. Pay attention to this point. The filters might have been considered part of the business logic and implemented in the use case itself, but we decided to leverage what the storage system can do, so we moved filtering in that external system. This is a reasonable choice as databases can usually perform filtering and ordering very well. Even though the in-memory storage we are currently using is not a database, we are preparing to use a real external storage.

The new version of repository tests is

tests/repository/test_memrepo.py

```
import pytest

from rentomatic.domain.room import Room
from rentomatic.repository.memrepo import MemRepo


@pytest.fixture
def room_dicts():
    return [
        {
```

```python
            "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,
        },
    ]


def test_repository_list_without_parameters(room_dicts):
    repo = MemRepo(room_dicts)

    rooms = [Room.from_dict(i) for i in room_dicts]
    assert repo.list() == rooms


def test_repository_list_with_code_equal_filter(room_dicts):
    repo = MemRepo(room_dicts)

    rooms = repo.list(
        filters={"code__eq": "fe2c3195-aeff-487a-a08f-e0bdc0
    )

    assert len(rooms) == 1
    assert rooms[0].code == "fe2c3195-aeff-487a-a08f-e0bdc0e


@pytest.mark.parametrize("price", [60, "60"])
def test_repository_list_with_price_equal_filter(room_dicts,
    repo = MemRepo(room_dicts)

    rooms = repo.list(filters={"price__eq": price})

    assert len(rooms) == 1
    assert rooms[0].code == "913694c6-435a-4366-ba0d-da5334a


@pytest.mark.parametrize("price", [60, "60"])
def test_repository_list_with_price_less_than_filter(room_di
    repo = MemRepo(room_dicts)

    rooms = repo.list(filters={"price__lt": price})

    assert len(rooms) == 2
    assert set([r.code for r in rooms]) == {
        "f853578c-fc0f-4e65-81b8-566c5dffa35a",
        "eed76e77-55c1-41ce-985d-ca49bf6c0585",
    }


@pytest.mark.parametrize("price", [48, "48"])
def test_repository_list_with_price_greater_than_filter(room
    repo = MemRepo(room_dicts)

    rooms = repo.list(filters={"price__gt": price})

    assert len(rooms) == 2
    assert set([r.code for r in rooms]) == {
        "913694c6-435a-4366-ba0d-da5334a611b2",
        "fe2c3195-aeff-487a-a08f-e0bdc0ec6e9a",
    }


def test_repository_list_with_price_between_filter(room_dict
    repo = MemRepo(room_dicts)

    rooms = repo.list(filters={"price__lt": 66, "price__gt":

    assert len(rooms) == 1
    assert rooms[0].code == "913694c6-435a-4366-ba0d-da5334a
```

As you can see, I added many tests. One test for each of the four accepted filters (`code__eq`, `price__eq`, `price__lt`, `price__gt`, see `rentomatic/requests/room_list.py`), and one final test that tries two different filters at the same time.

Again, keep in mind that this is the API exposed by the storage, not the one exposed by the use case. The fact that the two match is a design decision, but your mileage may vary.

The new version of the repository is

rentomatic/repository/memrepo.py

```
from rentomatic.domain.room import Room


class MemRepo:
    def __init__(self, data):
        self.data = data

    def list(self, filters=None):

        result = [Room.from_dict(i) for i in self.data]

        if filters is None:
            return result

        if "code__eq" in filters:
            result = [r for r in result if r.code == filters

        if "price__eq" in filters:
            result = [
                r for r in result if r.price == int(filters[
            ]

        if "price__lt" in filters:
            result = [
                r for r in result if r.price < int(filters["
            ]

        if "price__gt" in filters:
            result = [
                r for r in result if r.price > int(filters["
            ]

        return result
```

At this point, you can start the Flask development webserver with `FLASK_CONFIG="development" flask run`, and get the list of all your rooms at http://localhost:5000/rooms. You can also use filters in the URL, like http://localhost:5000/rooms?filter_code__eq=f853578c-fc0f-4e65-81b8-566c5dffa35a which returns the room with the given code or http://localhost:5000/rooms?filter_price__lt=50 which returns all the rooms with a price less than 50.

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c05-s08

The CLI

At this point fixing the CLI is extremely simple, as we just need to imitate what we did for the HTTP server, only without considering the filters as they were not part of the command line tool.

cli.py

```
#!/usr/bin/env python

from rentomatic.repository.memrepo import MemRepo
from rentomatic.use_cases.room_list import room_list_use_cas
from rentomatic.requests.room_list import build_room_list_re

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,
    },
]

request = build_room_list_request()
repo = MemRepo(rooms)
response = room_list_use_case(repo, request)
```

```
print([room.to_dict() for room in response.value])
```

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c05-s09

We now have a very robust system to manage input validation and error conditions, and it is generic enough to be used with any possible use case. Obviously, we are free to add new types of errors to increase the granularity with which we manage failures, but the present version already covers everything that can happen inside a use case.

In the next chapter, we will have a look at repositories based on real database engines, showing how to test external systems with integration tests, using PostgreSQL as a database. In a later chapter I will show how the clean architecture allows us to switch very easily between different external systems, moving the system to MongoDB.

1 For example using the browser developer tools. In Chrome and Firefox, press F12 and open the Network tab, then refresh the page.

Previous
Chapter 4 - Add a web application

Next
Chapter 6 - Integration with a real external system - Postgres

Last update: 20/08/2021

Share on: Twitter LinkedIn HackerNews Email Reddit

Section 7 of Clean Architectures in Python

Previous

Next