Clean Architectures in Python

Chapter 7 - Integration with a real external system - MongoDB

There's, uh, another example.

Jurassic Park, 1993

The previous chapter showed how to integrate a real external system with the core of the clean architecture. Unfortunately I also had to introduce a lot of code to manage the integration tests and to globally move forward to a proper setup. In this chapter I will leverage the work we just did to show only the part strictly connected with the external system. Swapping the database from PostgreSQL to MongoDB is the perfect way to show how flexible the clean architecture is, and how easy it is to introduce different approaches like a non-relational database instead of a relational one.

Fixtures ¶

Thanks to the flexibility of clean architecture, providing support for multiple storage systems is a breeze. In this section, I will implement the class `MongoRepo` that provides an interface towards MongoDB, a well-known NoSQL database. We will follow the same testing strategy we used for PostgreSQL, with a Docker container that runs the database and docker-compose that orchestrates the whole system.

You will appreciate the benefits of the complex testing structure that I created in the previous chapter. That structure allows me to reuse some of the fixtures now that I want to implement tests for a new storage system.

Let's start defining the file `tests/repository/mongodb/conftest.py`, which will contains pytest fixtures for MongoDB, mirroring the file we created for PostgreSQL

```
tests/repository/mongodb/conftest.py
```

```python
import pymongo
import pytest


@pytest.fixture(scope="session")
def mg_database_empty(app_configuration):
    client = pymongo.MongoClient(
        host=app_configuration["MONGODB_HOSTNAME"],
        port=int(app_configuration["MONGODB_PORT"]),
        username=app_configuration["MONGODB_USER"],
        password=app_configuration["MONGODB_PASSWORD"],
        authSource="admin",
    )
    db = client[app_configuration["APPLICATION_DB"]]

    yield db

    client.drop_database(app_configuration["APPLICATION_DB"]
    client.close()


@pytest.fixture(scope="function")
def mg_test_data():
    return [
        {
            "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,
        },
    ]


@pytest.fixture(scope="function")
def mg_database(mg_database_empty, mg_test_data):
    collection = mg_database_empty.rooms
    collection.insert_many(mg_test_data)

    yield mg_database_empty

    collection.delete_many({})
```

As you can see these functions are very similar to the ones that we defined for Postgres. The function `mg_database_empty` is tasked to create the MongoDB client and the empty database, and to dispose them after the `yield`. The fixture `mg_test_data` provides the same data provided by `pg_test_data` and `mg_database` fills the empty database with it. While the SQLAlchemy package works through a session, PyMongo library creates a client and uses it directly, but the overall structure is the same.

Since we are importing the PyMongo library we need to change the production requirements

requirements/prod.txt

```
Flask
SQLAlchemy
psycopg2
pymongo
```

and run `pip install -r requirements/dev.txt`.

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c07-s01

Docker Compose configuration ¶

We need to add an ephemeral MongoDB container to the testing Docker Compose configuration. The MongoDB image needs only the variables `MONGO_INITDB_ROOT_USERNAME` and `MONGO_INITDB_ROOT_PASSWORD` as it doesn't create any initial database. As we did for the PostgreSQL container we assign a specific port that will be different from the standard one, to allow tests to be executed while other containers are running.

docker/testing.yml

```yaml
version: '3.8'

services:
  postgres:
    image: postgres
    environment:
      POSTGRES_DB: ${POSTGRES_DB}
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
    ports:
      - "${POSTGRES_PORT}:5432"
  mongo:
    image: mongo
    environment:
      MONGO_INITDB_ROOT_USERNAME: ${MONGODB_USER}
      MONGO_INITDB_ROOT_PASSWORD: ${MONGODB_PASSWORD}
    ports:
      - "${MONGODB_PORT}:27017"
```

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c07-s02

Application configuration ¶

Docker Compose, the testing framework, and the application itself are configured through a single JSON file, that we need to update with the actual values we want to use for MongoDB

config/testing.json

```
[
  {
    "name": "FLASK_ENV",
    "value": "production"
  },
  {
    "name": "FLASK_CONFIG",
    "value": "testing"
  },
  {
    "name": "POSTGRES_DB",
    "value": "postgres"
  },
  {
    "name": "POSTGRES_USER",
    "value": "postgres"
  },
  {
    "name": "POSTGRES_HOSTNAME",
    "value": "localhost"
  },
  {
    "name": "POSTGRES_PORT",
    "value": "5433"
  },
  {
    "name": "POSTGRES_PASSWORD",
    "value": "postgres"
  },
  {
    "name": "MONGODB_USER",
    "value": "root"
  },
  {
    "name": "MONGODB_HOSTNAME",
    "value": "localhost"
  },
  {
    "name": "MONGODB_PORT",
    "value": "27018"
  },
  {
    "name": "MONGODB_PASSWORD",
    "value": "mongodb"
  },
  {
    "name": "APPLICATION_DB",
    "value": "test"
  }
]
```

Since the standard port from MongoDB is 27017 I chose 27018 for the tests. Remember that this is just an example, however. In a real scenario we might have multiple environments and also multiple setups for our testing, and in that case we might want to assign a random port to the container and use Python to extract the value and pass it to the application.

Please also note that I chose to use the same variable `APPLICATION_DB` for the name of the PostgreSQL and MongoDB databases. Again, this is a simple example, and your mileage my vary in more complex scenarios.

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c07-s03

Integration tests ¶

The integration tests are a mirror of the ones we wrote for Postgres, as we are covering the same use case. If you use multiple databases in the same system you probably want to serve different use cases, so in a real case this might probably be a more complicated step. It is completely reasonable, however, that you might want to simply provide support for multiple databases that your client can choose to plug into the system, and in that case you will do exactly what I did here, copying and adjusting the same test battery.

tests/repository/mongodb/test_mongorepo.py

```
import pytest
from rentomatic.repository import mongorepo

pytestmark = pytest.mark.integration


def test_repository_list_without_parameters(
    app_configuration, mg_database, mg_test_data
```

```
):
    repo = mongorepo.MongoRepo(app_configuration)

    repo_rooms = repo.list()

    assert set([r.code for r in repo_rooms]) == set(
        [r["code"] for r in mg_test_data]
    )


def test_repository_list_with_code_equal_filter(
    app_configuration, mg_database, mg_test_data
):
    repo = mongorepo.MongoRepo(app_configuration)

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

    assert len(repo_rooms) == 1
    assert repo_rooms[0].code == "fe2c3195-aeff-487a-a08f-e0


def test_repository_list_with_price_equal_filter(
    app_configuration, mg_database, mg_test_data
):
    repo = mongorepo.MongoRepo(app_configuration)

    repo_rooms = repo.list(filters={"price__eq": 60})

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

```
def test_repository_list_with_price_less_than_filter(
    app_configuration, mg_database, mg_test_data
):
    repo = mongorepo.MongoRepo(app_configuration)

    repo_rooms = repo.list(filters={"price__lt": 60})

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


def test_repository_list_with_price_greater_than_filter(
    app_configuration, mg_database, mg_test_data
):
    repo = mongorepo.MongoRepo(app_configuration)

    repo_rooms = repo.list(filters={"price__gt": 48})

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


def test_repository_list_with_price_between_filter(
    app_configuration, mg_database, mg_test_data
):
    repo = mongorepo.MongoRepo(app_configuration)

    repo_rooms = repo.list(filters={"price__lt": 66, "price_
    assert len(repo_rooms) == 1
    assert repo_rooms[0].code == "913694c6-435a-4366-ba0d-da


def test_repository_list_with_price_as_string(
    app_configuration, mg_database, mg_test_data
):
    repo = mongorepo.MongoRepo(app_configuration)

    repo_rooms = repo.list(filters={"price__lt": "60"})

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

I added a test called `test_repository_list_with_price_as_string` that checks what happens when the price in the filter is expressed as a string. Experimenting with the MongoDB shell I found that in this case the query wasn't working, so I included the test to be sure the implementation didn't forget to manage this condition.

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c07-s04

The MongoDB repository ¶

The `MongoRepo` class is obviously not the same as the Postgres interface, as the PyMongo library is different from SQLAlchemy, and the structure of a NoSQL database differs from the one of a relational one. The file `rentomatic/repository/mongorepo.py` is

rentomatic/repository/mongorepo.py

```
import pymongo

from rentomatic.domain import room


class MongoRepo:
    def __init__(self, configuration):
        client = pymongo.MongoClient(
            host=configuration["MONGODB_HOSTNAME"],
            port=int(configuration["MONGODB_PORT"]),
            username=configuration["MONGODB_USER"],
            password=configuration["MONGODB_PASSWORD"],
            authSource="admin",
        )

        self.db = client[configuration["APPLICATION_DB"]]

    def _create_room_objects(self, results):
        return [
            room.Room(
                code=q["code"],
                size=q["size"],
                price=q["price"],
                latitude=q["latitude"],
                longitude=q["longitude"],
            )
            for q in results
        ]

    def list(self, filters=None):
        collection = self.db.rooms

        if filters is None:
            result = collection.find()
        else:
            mongo_filter = {}
            for key, value in filters.items():
                key, operator = key.split("__")

                filter_value = mongo_filter.get(key, {})

                if key == "price":
                    value = int(value)

                filter_value["${}".format(operator)] = value
                mongo_filter[key] = filter_value

            result = collection.find(mongo_filter)

        return self._create_room_objects(result)
```

which makes use of the similarity between the filters of the Rent-o-matic project and the ones of the MongoDB systemfootnote:[The similitude between the two systems is not accidental, as I was studying MongoDB at the time I wrote the first article about clean architectures, so I was obviously influenced by it.].

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c07-s05

I think this very brief chapter clearly showed the merits of a layered approach and of a proper testing setup. So far we implemented and tested an interface towards two very different databases like PostgreSQL and MongoDB, but both interfaces are usable by the same use case, which ultimately means the same API endpoint.

While we properly tested the integration with these external systems, we still don't have a way to run the whole system in what we call a production-ready environment, that is in a way that can be exposed to external users. In the next chapter I will show you how we can leverage the same setup we used for the tests to run Flask, PostgreSQL, and the use case we created in a way that can be used in production.

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

Next
Chapter 8 - Run a production-ready system

Last update: 20/08/2021

Section 9 of Clean Architectures in Python

Previous

Next