Clean Architectures in Python

Chapter 6 - Integration with a real external system - Postgres

Ooooh, I'm very sorry Hans. I didn't get that memo.
Maybe you should've put it on the bulletin board.

Die Hard, 1988

The basic in-memory repository I implemented for the project is enough to show the concept of the repository layer abstraction. It is not enough to run a production system, though, so we need to implement a connection with a real storage like a database. Whenever we use an external system and we want to test the interface we can use mocks, but at a certain point we need to ensure that the two systems actually work together, and this is when we need to start creating integration tests.

In this chapter I will show how to set up and run integration tests between our application and a real database. At the end of the chapter I will have a repository that allows the application to interface with PostgreSQL, and a battery of tests that run using a real database instance running in Docker.

This chapter will show you one of the biggest advantages of a clean architecture, namely the simplicity with which you can replace existing components with others, possibly based on a completely different technology.

Decoupling with interfaces ¶

The clean architecture we devised in the previous chapters defines a use case that receives a repository instance as an argument and uses its `list` method to retrieve the contained entries. This allows the use case to form a very loose coupling with the repository, being connected only through the API exposed by the object and not to the real implementation. In other words, the use cases are polymorphic with respect to the method `list`.

This is very important and it is the core of the clean architecture design. Being connected through an API, the use case and the repository can be replaced by different implementations at any time, given that the new implementation provides the requested interface.

It is worth noting, for example, that the initialisation of the object is not part of the API that the use cases are using since the repository is initialised in the main script and not in each use case. The method `__init__`, thus, doesn't need to be the same among the repository implementations, which gives us a great deal of flexibility, as different storage systems may need different initialisation values.

The simple repository we implemented in one of the previous chapters 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
```

whose interface is made of two parts: the initialisation and the method `list`. The method `__init__` accepts values because this specific object doesn't act as long-term storage, so we are forced to pass some data every time we instantiate the class.

A repository based on a proper database will not need to be filled with data when initialised, its main job being that of storing data between sessions, but will nevertheless need to be initialised at least with the database address and access credentials.

Furthermore, we have to deal with a proper external system, so we have to devise a strategy to test it, as this might require a running database engine in the

background. Remember that we are creating a specific implementation of a repository, so everything will be tailored to the actual database system that we will choose.

A repository based on PostgreSQL ¶

Let's start with a repository based on a popular SQL database, PostgreSQL. It can be accessed from Python in many ways, but the best one is probably through the SQLAlchemy interface. SQLAlchemy is an ORM, a package that maps objects (as in object-oriented) to a relational database. ORMs can normally be found in web frameworks like Django or in standalone packages like the one we are considering.

The important thing about ORMs is that they are very good examples of something you shouldn't try to mock. Properly mocking the SQLAlchemy structures that are used when querying the DB results in very complex code that is difficult to write and almost impossible to maintain, as every single change in the queries results in a series of mocks that have to be written again[1].

We need therefore to set up an integration test. The idea is to create the DB, set up the connection with SQLAlchemy, test the condition we need to check, and destroy the database. Since the action of creating and destroying the DB can be expensive in terms of time, we might want to do it just at the beginning and at the end of the whole test suite, but even with this change, the tests will be slow. This is why we will also need to use labels to avoid running them every time we run the suite. Let's face this complex task one step at a time.

Label integration tests ¶

The first thing we need to do is to label integration tests, exclude them by default and create a way to run them. Since pytest supports labels, called *marks*, we can use this feature to add a global mark to a whole module. Create the file

`tests/repository/postgres/test_postgresrepo.py` and put in it this code

tests/repository/postgres/test_postgresrepo.py

```
import pytest

pytestmark = pytest.mark.integration


def test_dummy():
    pass
```

The module attribute `pytestmark` labels every test in the module with the tag `integration`. To verify that this works I added a `test_dummy` test function which always passes.

The marker should be registered in `pytest.ini`

pytest.ini

```
[pytest]
minversion = 2.0
norecursedirs = .git .tox requirements*
python_files = test*.py
markers =
        integration: integration tests
```

You can now run `pytest -svv -m integration` to ask pytest to run only the tests marked with that label. The option `-m` supports a rich syntax that you can learn by reading the documentation.

```
$ pytest -svv -m integration
========================== test session starts ==========
```

```
platform linux -- Python XXXX, pytest-XXXX, py-XXXX, pluggy-
cabook/venv3/bin/python3
cachedir: .cache
rootdir: cabook/code/calc, inifile: pytest.ini
plugins: cov-XXXX
collected 36 items / 35 deselected / 1 selected

tests/repository/postgres/test_postgresrepo.py::test_dummy P

=================== 1 passed, 35 deselected in 0.20s =======
```

While this is enough to run integration tests selectively, it is not enough to skip them by default. To do this, we can alter the pytest setup to label all those tests as skipped, but this will give us no means to run them. The standard way to implement this is to define a new command-line option and to process each marked test according to the value of this option.

To do it open the file `tests/conftest.py` that we already created and add the following code

tests/conftest.py

```
def pytest_addoption(parser):
    parser.addoption(
        "--integration", action="store_true", help="run inte
    )


def pytest_runtest_setup(item):
    if "integration" in item.keywords and not item.config.ge
        "integration"
```

```
    ):
        pytest.skip("need --integration option to run")
```

The first function is a hook into the pytest CLI parser that adds the option `--integration`. When this option is specified on the command line the pytest setup will contain the key `integration` with value `True`.

The second function is a hook into the pytest setup of every single test. The variable `item` contains the test itself (actually a `_pytest.python.Function` object), which in turn contains two useful pieces of information. The first is the attribute `item.keywords`, that contains the test marks, alongside many other interesting things like the name of the test, the file, the module, and also information about the patches that happen inside the test. The second is the attribute `item.config` that contains the parsed pytest command line.

So, if the test is marked with `integration` (`'integration' in item.keywords`) and the option `--integration` is not present (`not item.config.getvalue("integration")`) the test is skipped.

This is the output with `--integration`

```
$ pytest -svv --integration
========================== test session starts ==============
platform linux -- Python XXXX, pytest-XXXX, py-XXXX, pluggy-
cabook/venv3/bin/python3
cachedir: .cache
rootdir: cabook/code/calc, inifile: pytest.ini
plugins: cov-XXXX
collected 36 items

...
tests/repository/postgres/test_postgresrepo.py::test_dummy P
...
```

```
========================= 36 passed in 0.26s ==============
```

and this is the output without the custom option

```
$ pytest -svv
========================= test session starts =============
platform linux -- Python XXXX, pytest-XXXX, py-XXXX, pluggy-
cabook/venv3/bin/python3
cachedir: .cache
rootdir: cabook/code/calc, inifile: pytest.ini
plugins: cov-XXXX
collected 36 items

...
tests/repository/postgres/test_postgresrepo.py::test_dummy S
...

=================== 35 passed, 1 skipped in 0.27s ==========
```

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c06-s01

Create SQLAlchemy classes ¶

Creating and populating the test database with initial data will be part of the test suite, but we need to define somewhere the tables that will be contained

in the database. This is where SQLAlchemy's ORM comes into play, as we will define those tables in terms of Python objects.

Add the packages `SQLAlchemy` and `psycopg2` to the requirements file `prod.txt`

requirements/prod.txt

```
Flask
SQLAlchemy
psycopg2
```

and update the installed packages with

```
$ pip install -r requirements/dev.txt
```

Create the file `rentomatic/repository/postgres_objects.py` with the following content

rentomatic/repository/postgres_objects.py

```
from sqlalchemy import Column, Integer, String, Float
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()


class Room(Base):
    __tablename__ = 'room'

    id = Column(Integer, primary_key=True)

    code = Column(String(36), nullable=False)
    size = Column(Integer)
```

```
    price = Column(Integer)
    longitude = Column(Float)
    latitude = Column(Float)
```

Let's comment it section by section

```
from sqlalchemy import Column, Integer, String, Float
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
```

We need to import many things from the SQLAlchemy package to set up the database and to create the table. Remember that SQLAlchemy has a declarative approach, so we need to instantiate the object `Base` and then use it as a starting point to declare the tables/objects.

```
class Room(Base):
    __tablename__ = 'room'

    id = Column(Integer, primary_key=True)

    code = Column(String(36), nullable=False)
    size = Column(Integer)
    price = Column(Integer)
    longitude = Column(Float)
    latitude = Column(Float)
```

This is the class that represents the room in the database. It is important to understand that this is not the class we are using in the business logic, but the class that defines the table in the SQL database that we will use to map the `Room` entity. The structure of this class is thus dictated by the needs of the storage layer, and not by the use cases. You might want for instance to store `lon-`

`gitude` and `latitude` in a JSON field, to allow for easier extendibility, without changing the definition of the domain model. In the simple case of the Rent-o-matic project, the two classes almost overlap, but this is not the case generally speaking.

Obviously, this means that you have to keep the storage and the domain levels in sync and that you need to manage migrations on your own. You can use tools like Alembic, but the migrations will not come directly from domain model changes.

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c06-s02

Orchestration management ¶

When we run the integration tests the Postgres database engine must be already running in the background, and it must be already configured, for example, with a pristine database ready to be used. Moreover, when all the tests have been executed the database should be removed and the database engine stopped.

This is a perfect job for Docker, which can run complex systems in isolation with minimal configuration. We have a choice here: we might want to orchestrate the creation and destruction of the database with an external script or try to implement everything in the test suite. The first solution is what many frameworks use, and what I explored in my series of posts Flask Project Setup: TDD, Docker, Postgres and more, so in this chapter I will show an implementation of that solution.

As I explained in the posts I mentioned the plan is to create a management script that spins up and tears down the required containers, runs the tests in between. The management script can be used also to run the application itself,

or to create development setups, but in this case I will simplify it to manage only the tests. I highly recommend that you read those posts if you want to get the big picture behind the setup I will use.

The first thing we have to do if we plan to use Docker Compose is to add the requirement to `requirements/test.txt`

requirements/test.txt

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

and install it running `pip install -r requirements/dev.txt`. The management script is the following

manage.py

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

import os
import json
import subprocess
import time

import click
import psycopg2
from psycopg2.extensions import ISOLATION_LEVEL_AUTOCOMMIT


# Ensure an environment variable exists and has a value
```

```
def setenv(variable, default):
    os.environ[variable] = os.getenv(variable, default)


APPLICATION_CONFIG_PATH = "config"
DOCKER_PATH = "docker"


def app_config_file(config):
    return os.path.join(APPLICATION_CONFIG_PATH, f"{config}.


def docker_compose_file(config):
    return os.path.join(DOCKER_PATH, f"{config}.yml")


def read_json_configuration(config):
    # Read configuration from the relative JSON file
    with open(app_config_file(config)) as f:
        config_data = json.load(f)

    # Convert the config into a usable Python dictionary
    config_data = dict((i["name"], i["value"]) for i in conf

    return config_data


def configure_app(config):
    configuration = read_json_configuration(config)

    for key, value in configuration.items():
        setenv(key, value)
```

```
@click.group()
def cli():
    pass


def docker_compose_cmdline(commands_string=None):
    config = os.getenv("APPLICATION_CONFIG")
    configure_app(config)

    compose_file = docker_compose_file(config)

    if not os.path.isfile(compose_file):
        raise ValueError(f"The file {compose_file} does not

    command_line = [
        "docker-compose",
        "-p",
        config,
        "-f",
        compose_file,
    ]

    if commands_string:
        command_line.extend(commands_string.split(" "))

    return command_line


def run_sql(statements):
    conn = psycopg2.connect(
        dbname=os.getenv("POSTGRES_DB"),
        user=os.getenv("POSTGRES_USER"),
        password=os.getenv("POSTGRES_PASSWORD"),
        host=os.getenv("POSTGRES_HOSTNAME"),
```

```
        port=os.getenv("POSTGRES_PORT"),
    )

    conn.set_isolation_level(ISOLATION_LEVEL_AUTOCOMMIT)
    cursor = conn.cursor()
    for statement in statements:
        cursor.execute(statement)

    cursor.close()
    conn.close()


def wait_for_logs(cmdline, message):
    logs = subprocess.check_output(cmdline)
    while message not in logs.decode("utf-8"):
        time.sleep(1)
        logs = subprocess.check_output(cmdline)


@cli.command()
@click.argument("args", nargs=-1)
def test(args):
    os.environ["APPLICATION_CONFIG"] = "testing"
    configure_app(os.getenv("APPLICATION_CONFIG"))

    cmdline = docker_compose_cmdline("up -d")
    subprocess.call(cmdline)

    cmdline = docker_compose_cmdline("logs postgres")
    wait_for_logs(cmdline, "ready to accept connections")

    run_sql([f"CREATE DATABASE {os.getenv('APPLICATION_DB')}

    cmdline = [
```

```
        "pytest",
        "-svv",
        "--cov=application",
        "--cov-report=term-missing",
    ]
    cmdline.extend(args)
    subprocess.call(cmdline)

    cmdline = docker_compose_cmdline("down")
    subprocess.call(cmdline)


if __name__ == "__main__":
    cli()
```

Let's see what it does block by block.

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

import os
import json
import subprocess
import time

import click
import psycopg2
from psycopg2.extensions import ISOLATION_LEVEL_AUTOCOMMIT


# Ensure an environment variable exists and has a value
def setenv(variable, default):
    os.environ[variable] = os.getenv(variable, default)
```

```
APPLICATION_CONFIG_PATH = "config"
DOCKER_PATH = "docker"
```

Some Docker containers (like the PostgreSQL one that we will use shortly) depend on environment variables to perform the initial setup, so we need to define a function to set environment variables if they are not already initialised. We also define a couple of paths for configuration files.

```
def app_config_file(config):
    return os.path.join(APPLICATION_CONFIG_PATH, f"{config}.


def docker_compose_file(config):
    return os.path.join(DOCKER_PATH, f"{config}.yml")


def read_json_configuration(config):
    # Read configuration from the relative JSON file
    with open(app_config_file(config)) as f:
        config_data = json.load(f)

    # Convert the config into a usable Python dictionary
    config_data = dict((i["name"], i["value"]) for i in conf

    return config_data


def configure_app(config):
    configuration = read_json_configuration(config)
```

```
    for key, value in configuration.items():
        setenv(key, value)
```

As in principle I expect to have a different configuration at least for development, testing, and production, I introduced `app_config_file` and `docker_compose_file` that return the specific file for the environment we are working in. The function `read_json_configuration` has been isolated from `configure_app` as it will be imported by the tests to initialise the database repository.

```
@click.group()
def cli():
    pass


def docker_compose_cmdline(commands_string=None):
    config = os.getenv("APPLICATION_CONFIG")
    configure_app(config)

    compose_file = docker_compose_file(config)

    if not os.path.isfile(compose_file):
        raise ValueError(f"The file {compose_file} does not

    command_line = [
        "docker-compose",
        "-p",
        config,
        "-f",
        compose_file,
    ]
```

```
    if commands_string:
        command_line.extend(commands_string.split(" "))

    return command_line
```

This is a simple function that creates the Docker Compose command line that avoids repeating long lists of options whenever we need to orchestrate the containers.

```
def run_sql(statements):
    conn = psycopg2.connect(
        dbname=os.getenv("POSTGRES_DB"),
        user=os.getenv("POSTGRES_USER"),
        password=os.getenv("POSTGRES_PASSWORD"),
        host=os.getenv("POSTGRES_HOSTNAME"),
        port=os.getenv("POSTGRES_PORT"),
    )

    conn.set_isolation_level(ISOLATION_LEVEL_AUTOCOMMIT)
    cursor = conn.cursor()
    for statement in statements:
        cursor.execute(statement)

    cursor.close()
    conn.close()


def wait_for_logs(cmdline, message):
    logs = subprocess.check_output(cmdline)
    while message not in logs.decode("utf-8"):
        time.sleep(1)
        logs = subprocess.check_output(cmdline)
```

The function `run_sql` allows us to run SQL commands on a running Postgres database, and will come in handy when we will create the empty test database. The second function, `wait_for_logs` is a simple way to monitor the Postgres container and to be sure it's ready to be used. Whenever you spin up containers programmatically you need to be aware that they have a certain startup time before they are ready, and act accordingly.

```
@cli.command()
@click.argument("args", nargs=-1)
def test(args):
    os.environ["APPLICATION_CONFIG"] = "testing"
    configure_app(os.getenv("APPLICATION_CONFIG"))

    cmdline = docker_compose_cmdline("up -d")
    subprocess.call(cmdline)

    cmdline = docker_compose_cmdline("logs postgres")
    wait_for_logs(cmdline, "ready to accept connections")

    run_sql([f"CREATE DATABASE {os.getenv('APPLICATION_DB')}

    cmdline = [
        "pytest",
        "-svv",
        "--cov=application",
        "--cov-report=term-missing",
    ]
    cmdline.extend(args)
    subprocess.call(cmdline)

    cmdline = docker_compose_cmdline("down")
    subprocess.call(cmdline)
```

```
if __name__ == "__main__":
    cli()
```

This function is the last that we define, and the only command provided by our management script. First of all the application is configured with the name `testing`, which means that we will use the configuration file `config/testing.json` and the Docker Compose file `docker/testing.yml`. All these names and paths are just conventions that comes from the arbitrary setup of this management script, so you are clearly free to structure your project in a different way.

The function then spins up the containers according to the Docker Compose file, running `docker-compose up -d`. It waits for the log message that communicates the database is ready to accept connections and runs the SQL command that creates the testing database.

After this it runs Pytest with a default set of options, adding all the options that we will provide on the command line, and eventually tears down the Docker Compose containers.

To complete the setup we need to define a configuration file for Docker Compose

docker/testing.yml

```
version: '3.8'

services:
  postgres:
    image: postgres
    environment:
      POSTGRES_DB: ${POSTGRES_DB}
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
```

```
    ports:
      - "${POSTGRES_PORT}:5432"
```

And finally a JSON configuration file

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": "APPLICATION_DB",
    "value": "test"
  }
]
```

A couple of notes about this configuration. First of all it defines both `FLASK_ENV` and `FLASK_CONFIG`. The first is, as you might remember, and internal Flask variable that can only be `development` or `production`, and is connected with the internal debugger. The second is the variable that we use to configure our Flask application with the objects in `application/config.py`. For testing purposes we set `FLASK_ENV` to `production` as we don't need the internal debugger, and `FLASK_CONFIG` to `testing`, which will resul in the application being configured with the class `TestingConfig`. This class sets the internal Flask parameter `TESTING` to `True`.

The rest of the JSON configuration initialises variables whose names start with the prefix `POSTGRES_`. These are variables required by the Postgres Docker container. When the container is run, it automatically creates a database with the name specified by `POSTGRES_DB`. It also creates a user with a password, using the values specified in `POSTGRES_USER` and `POSTGRES_PASSWORD`.

Last, I introduced the variable `APPLICATION_DB` because I want to create a specific database which is not the one the default one. The default port `POSTGRES_PORT` has been changed from the standard value 5432 to 5433 to avoid clashing with any database already running on the machine (either natively or containerised). As you can see in the Docker Compose configuration file this changes only the external mapping of the container and not the actual port the database engine is using inside the container.

With all these files in place we are ready to start designing our tests.

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c06-s03

Database fixtures ¶

As we defined the configuration of the database in a JSON file we need a fixture that loads that same configuration, so that we can connect to the database during the tests. As we already have the function `read_json_configuration` in the management script we just need to wrap that. This is a fixture that is not specific to the Postgres repository, so I will introduce it in `tests/conftest.py`

tests/conftest.py

```
from manage import read_json_configuration

...

@pytest.fixture(scope="session")
def app_configuration():
    return read_json_configuration("testing")
```

As you can see I hardcoded the name of the configuration file for simplicity's sake. Another solution might be to create an environment variable with the application configuration in the management script and to read it from here.

The rest of the fixtures contains code that is specific to Postgres, so it is better to keep the code separated in a more specific file `conftest.py`

tests/repository/postgres/conftest.py

```
import sqlalchemy
import pytest
```

```
from rentomatic.repository.postgres_objects import Base, Roo


@pytest.fixture(scope="session")
def pg_session_empty(app_configuration):
    conn_str = "postgresql+psycopg2://{}:{}@{}:{}/{}".format
        app_configuration["POSTGRES_USER"],
        app_configuration["POSTGRES_PASSWORD"],
        app_configuration["POSTGRES_HOSTNAME"],
        app_configuration["POSTGRES_PORT"],
        app_configuration["APPLICATION_DB"],
    )
    engine = sqlalchemy.create_engine(conn_str)
    connection = engine.connect()

    Base.metadata.create_all(engine)
    Base.metadata.bind = engine

    DBSession = sqlalchemy.orm.sessionmaker(bind=engine)
    session = DBSession()

    yield session

    session.close()
    connection.close


@pytest.fixture(scope="session")
def pg_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 pg_session(pg_session_empty, pg_test_data):
    for r in pg_test_data:
        new_room = Room(
            code=r["code"],
            size=r["size"],
```

```
            price=r["price"],
            longitude=r["longitude"],
            latitude=r["latitude"],
        )
        pg_session_empty.add(new_room)
        pg_session_empty.commit()

    yield pg_session_empty

    pg_session_empty.query(Room).delete()
```

The first fixture `pg_session_empty` creates a session to the empty initial database, while `pg_test_data` defines the values that we will load into the database. As we are not mutating this set of values we don't need to create a fixture, but this is the easier way to make it available both to the other fixtures and to the tests. The last fixture `pg_session` fills the database with Postgres objects created with the test data. Pay attention that these are not entities, but the Postgres objects we created to map them.

Note that this last fixture has a `function` scope, thus it is run for every test. Therefore, we delete all rooms after the yield returns, leaving the database exactly as it was before the test. Generally speaking you should always clean up after tests. The endpoint we are testing does not write to the database so in this specific case there is no real need to clean up, but I prefer to implement a complete solution from step zero.

We can test this whole setup changing the function `test_dummy` so that it fetches all the rows of the table `Room` and verifying that the query returns 4 values.

The new version of `tests/repository/postgres/test_postgresrepo.py` is

```
import pytest
from rentomatic.repository.postgres_objects import Room

pytestmark = pytest.mark.integration


def test_dummy(pg_session):
    assert len(pg_session.query(Room).all()) == 4
```

At this point you can run the test suite with integration tests. You should notice a clear delay when pytest executes the function `test_dummy` as Docker will take some time to spin up the database container and prepare the data

```
$ ./manage.py test -- --integration
========================== test session starts ==============
platform linux -- Python XXXX, pytest-XXXX, py-XXXX, pluggy-
cabook/venv3/bin/python3
cachedir: .cache
rootdir: cabook/code/calc, inifile: pytest.ini
plugins: cov-XXXX
collected 36 items

...
tests/repository/postgres/test_postgresrepo.py::test_dummy P
...

========================== 36 passed in 0.26s ===============
```

Note that to pass the option `--integration` we need to use `--` otherwise Click would consider the option as belonging to the script `./manage.py` instead of passing it as a pytest argument.

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c06-s04

Integration tests ¶

At this point we can create the real tests in the file `test_postgresrepo.py`, replacing the function `test_dummy`. All test receive the fixtures `app_configuration`, `pg_session`, and `pg_test_data`. The first fixture allows us to initialise the class `PostgresRepo` using the proper parameters. The second creates the database using the test data that is then contained in the third fixture.

The tests for this repository are basically a copy of the ones created for `MemRepo`, which is not surprising. Usually, you want to test the very same conditions, whatever the storage system. Towards the end of the chapter we will see, however, that while these files are initially the same, they can evolve differently as we find bugs or corner cases that come from the specific implementation (in-memory storage, PostgreSQL, and so on).

tests/repository/postgres/test_postgresrepo.py

```
import pytest
from rentomatic.repository import postgresrepo

pytestmark = pytest.mark.integration


def test_repository_list_without_parameters(
    app_configuration, pg_session, pg_test_data
):
    repo = postgresrepo.PostgresRepo(app_configuration)
```

```
    repo_rooms = repo.list()

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


def test_repository_list_with_code_equal_filter(
    app_configuration, pg_session, pg_test_data
):
    repo = postgresrepo.PostgresRepo(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, pg_session, pg_test_data
):
    repo = postgresrepo.PostgresRepo(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, pg_session, pg_test_data
```

```
):
    repo = postgresrepo.PostgresRepo(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, pg_session, pg_test_data
):
    repo = postgresrepo.PostgresRepo(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, pg_session, pg_test_data
):
    repo = postgresrepo.PostgresRepo(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
```

Remember that I introduced these tests one at a time and that I'm not showing you the full TDD workflow only for brevity's sake. The code of the class `PostgresRepo` has been developed following a strict TDD approach, and I recommend you to do the same. The resulting code goes in `rentomatic/repository/postgresrepo.py`, the same directory where we created the file `postgres_objects.py`.

rentomatic/repository/postgresrepo.py

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

from rentomatic.domain import room
from rentomatic.repository.postgres_objects import Base, Roo


class PostgresRepo:
    def __init__(self, configuration):
        connection_string = "postgresql+psycopg2://{}:{}@{}:
            configuration["POSTGRES_USER"],
            configuration["POSTGRES_PASSWORD"],
            configuration["POSTGRES_HOSTNAME"],
            configuration["POSTGRES_PORT"],
            configuration["APPLICATION_DB"],
        )

        self.engine = create_engine(connection_string)
        Base.metadata.create_all(self.engine)
        Base.metadata.bind = self.engine
```

```
    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):
        DBSession = sessionmaker(bind=self.engine)
        session = DBSession()

        query = session.query(Room)

        if filters is None:
            return self._create_room_objects(query.all())

        if "code__eq" in filters:
            query = query.filter(Room.code == filters["code_

        if "price__eq" in filters:
            query = query.filter(Room.price == filters["pric

        if "price__lt" in filters:
            query = query.filter(Room.price < filters["price

        if "price__gt" in filters:
            query = query.filter(Room.price > filters["price
```

```
        return self._create_room_objects(query.all())
```

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c06-s05

You might notice that `PostgresRepo` is very similar to `MemRepo`. This is the case because the case we are dealing with here, the list of `Room` objects, is pretty simple, so I don't expect great differences between an in-memory database an a production-ready relational one. As the use cases get more complex you will need to start leveraging the features provided by the engine that you are using, and methods such as `list` might evolve to become very different.

Note that the method `list` returns domain models, which is allowed as the repository is implemented in one of the outer layers of the architecture.

As you can see, while setting up a proper integration testing environment is not trivial, the changes that our architecture required to work with a real repository are very limited. I think this is a good demonstration of the flexibility of a layered approach such as the one at the core of the clean architecture.

Since this chapter mixed the setup of the integration testing with the introduction of a new repository, I will dedicate the next chapter purely to introduce a repository based on MongoDB, using the same structure that I created in this chapter. Supporting multiple databases (in this case even relational and non-relational) is not an uncommon pattern, as it allows you to use the approach that best suits each use case.

1 Unless you consider things like `sessionmaker_mock()().query.assert_called_with(Room)` something attractive. And this was by far the simplest mock I had to write.

Previous
Chapter 5 - Error management

Next
Chapter 7 - Integration with a real external system - MongoDB

Last update: 20/08/2021

Section 8 of Clean Architectures in Python

Previous

Next