Clean Architectures in Python

Chapter 8 - Run a production-ready system

Vilos Cohaagen said troops would be used to ensure full production.

Total Recall, 1990

Now that we developed a repository that connects with PostgreSQL we can discuss how to properly set up the application to run a production-ready system. This part is not strictly related to the clean architecture, but I think it's worth completing the example, showing how the system that we designed can end up being the core of a real web application.

Clearly, the definition "production-ready" refers to many different configuration that ultimately depend on the load and the business requirements of the system. As the goal is to show a complete example and not to cover real production requirements I will show a solution that uses real external systems like PostgreSQL and Nginx, without being too concerned about performances.

Build a web stack ¶

Now that we successfully containerised the tests we might try to devise a production-ready setup of the whole application, running both a web server and a database in Docker containers. Once again, I will follow the approach that I show in the series of posts I mentioned in one of the previous sections.

To run a production-ready infrastructure we need to put a WSGI server in front of the web framework and a Web server in front of it. We will also need to run a database container that we will initialise only once.

The steps towards a production-ready configuration are not complicated and the final setup won't be ultimately too different form what we already did for the tests. We need to

1. Create a JSON configuration with environment variables suitable for production
2. Create a suitable configuration for Docker Compose and configure the containers
3. Add commands to `manage.py` that allow us to control the processes

Let's create the file `config/production.json`, which is very similar to the one we created for the tests

config/production.json

```
[
  {
    "name": "FLASK_ENV",
    "value": "production"
  },
  {
    "name": "FLASK_CONFIG",
    "value": "production"
  },
  {
    "name": "POSTGRES_DB",
    "value": "postgres"
  },
  {
    "name": "POSTGRES_USER",
    "value": "postgres"
  },
```

```
    {
      "name": "POSTGRES_HOSTNAME",
      "value": "localhost"
    },
    {
      "name": "POSTGRES_PORT",
      "value": "5432"
    },
    {
      "name": "POSTGRES_PASSWORD",
      "value": "postgres"
    },
    {
      "name": "APPLICATION_DB",
      "value": "application"
    }
]
```

Please note that now both `FLASK_ENV` and `FLASK_CONFIG` are set to `production`. Please remember that the first is an internal Flask variable with two possible fixed values (`development` and `production`), while the second one is an arbitrary name that has the final effect of loading a specific configuration object (`ProductionConfig` in this case). I also changed `POSTGRES_PORT` back to the default `5432` and `APPLICATION_DB` to `application` (an arbitrary name).

Let's define which containers we want to run in our production environment, and how we want to connect them. We need a production-ready database and I will use Postgres, as I already did during the tests. Then we need to wrap Flask with a production HTTP server, and for this job I will use `gunicorn`. Last, we need a Web Server to act as load balancer.

The file `docker/production.yml` will contain the Docker Compose configuration, according to the convention we defined in `manage.py`

docker/production.yml

```yaml
version: '3.8'

services:
  db:
    image: postgres
    environment:
      POSTGRES_DB: ${POSTGRES_DB}
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
    ports:
      - "${POSTGRES_PORT}:5432"
    volumes:
      - pgdata:/var/lib/postgresql/data
  web:
    build:
      context: ${PWD}
      dockerfile: docker/web/Dockerfile.production
    environment:
      FLASK_ENV: ${FLASK_ENV}
      FLASK_CONFIG: ${FLASK_CONFIG}
      APPLICATION_DB: ${APPLICATION_DB}
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_HOSTNAME: "db"
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_PORT: ${POSTGRES_PORT}
    command: gunicorn -w 4 -b 0.0.0.0 wsgi:app
    volumes:
      - ${PWD}:/opt/code
  nginx:
    image: nginx
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
```

```
    ports:
      - 8080:8080

volumes:
  pgdata:
```

As you can see the Postgres configuration is not different from the one we used in the file `testing.yml`, but I added the option `volumes` (both in `db` and at the end of the file) that allows me to create a stable volume. If you don't do it, the database will be destroyed once you shut down the container.

The container `web` runs the Flask application through `gunicorn`. The environment variables come once again from the JSON configuration, and we need to define them because the application needs to know how to connect with the database and how to run the web framework. The command `gunicorn -w 4 -b 0.0.0.0 wsgi:app` loads the WSGI application we created in `wsgi.py` and runs it in 4 concurrent processes. This container is created using `docker/web/Dockerfile.production` which I still have to define.

The last container is `nginx`, which we will use as it is directly from the Docker Hub. The container runs Nginx with the configuration stored in `/etc/nginx/nginx.conf`, which is the file we overwrite with the local one `./nginx/nginx.conf`. Please note that I configured it to use port 8080 instead of the standard port 80 for HTTP to avoid clashing with other software that you might be running on your computer.

The Dockerfile for the web application is the following

docker/web/Dockerfile.production

```
FROM python:3

ENV PYTHONUNBUFFERED 1

RUN mkdir /opt/code
```

```
RUN mkdir /opt/requirements
WORKDIR /opt/code

ADD requirements /opt/requirements
RUN pip install -r /opt/requirements/prod.txt
```

This is a very simple container that uses the standard `python:3` image, where I added the production requirements contained in `requirements/prod.txt`. To make the Docker container work we need to add `gunicorn` to this last file

requirements/prod.txt

```
Flask
SQLAlchemy
psycopg2
pymongo
gunicorn
```

The configuration for Nginx is

docker/nginx/nginx.conf

```
worker_processes 1;

events { worker_connections 1024; }

http {

    sendfile on;

    upstream app {
        server web:8000;
    }
```

```
    server {
        listen 8080;

        location / {
            proxy_pass          http://app;
            proxy_redirect      off;
            proxy_set_header    Host $host;
            proxy_set_header    X-Real-IP $remote_addr;
            proxy_set_header    X-Forwarded-For $proxy_add_x_
            proxy_set_header    X-Forwarded-Host $server_name
        }
    }
}
```

As for the rest of the project, this configuration is very basic and lacks some important parts that are mandatory in a real production environment, such as HTTPS. In its essence, though, it is however not too different from the configuration of a production-ready Nginx container.

As we will use Docker Compose, the script `manage.py` needs a simple change, which is a command that wraps `docker-compose` itself. We need the script to just initialise environment variables according to the content of the JSON configuration file and then run Docker Compose. As we already have the function `docker_compose_cmdline` the job is pretty simple

manage.py

```
# Ensure an environment variable exists and has a value
import os
import json
import signal
import subprocess
import time
```

```
...

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


setenv("APPLICATION_CONFIG", "production")

APPLICATION_CONFIG_PATH = "config"
DOCKER_PATH = "docker"

...

@cli.command(context_settings={"ignore_unknown_options": Tru
@click.argument("subcommand", nargs=-1, type=click.Path())
def compose(subcommand):
    configure_app(os.getenv("APPLICATION_CONFIG"))
    cmdline = docker_compose_cmdline() + list(subcommand)

    try:
        p = subprocess.Popen(cmdline)
        p.wait()
    except KeyboardInterrupt:
        p.send_signal(signal.SIGINT)
        p.wait()
```

As you can see I forced the variable `APPLICATION_CONFIG` to be `production` if not specified. Usually, my default configuration is the development one, but in this simple case I haven't defined one, so this will do for now.

The new command is `compose`, that leverages Click's `argument` decorator to collect subcommands and attach them to the Docker Compose command line.

I also use the `signal` library, which I added to the imports, to control keyboard interruptions.

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c08-s01

When all this changes are in place we can test the application Dockerfile building the container.

```
$ ./manage.py compose build web
```

This command runs the Click command `compose` that first reads environment variables from the file `config/production.json`, and then runs `docker-compose` passing it the subcommand `build web`.

You output should be the following (with different image IDs)

```
Building web
Step 1/7 : FROM python:3
 ---> 768307cdb962
Step 2/7 : ENV PYTHONUNBUFFERED 1
 ---> Using cache
 ---> 0f2bb60286d3
Step 3/7 : RUN mkdir /opt/code
 ---> Using cache
 ---> e1278ef74291
Step 4/7 : RUN mkdir /opt/requirements
 ---> Using cache
 ---> 6d23f8abf0eb
Step 5/7 : WORKDIR /opt/code
 ---> Using cache
```

```
 ---> 8a3b6ae6d21c
Step 6/7 : ADD requirements /opt/requirements
 ---> Using cache
 ---> 75133f765531
Step 7/7 : RUN pip install -r /opt/requirements/prod.txt
 ---> Using cache
 ---> db644df9ba04

Successfully built db644df9ba04
Successfully tagged production_web:latest
```

If this is successful you can run Docker Compose

```
$ ./manage.py compose up -d
Creating production_web_1   ... done
Creating production_db_1    ... done
Creating production_nginx_1 ... done
```

and the output of `docker ps` should show three containers running

```
$ docker ps
IMAGE          PORTS                                   NAMES
nginx          80/tcp, 0.0.0.0:8080->8080/tcp          production_n
postgres       0.0.0.0:5432->5432/tcp                  production_d
production_web                                         production_w
```

Note that I removed several columns to make the output readable.

At this point we can open http://localhost:8080/rooms with our browser and see the result of the HTTP request received by Nginx, passed to gunicorn, and processed by Flask using the use case `room_list_use_case`.

The application is not actually using the database yet, as the Flask endpoint `room_list` in `application/rest/room.py` initialises the class `MemRepo` and loads it with some static values, which are the ones we see in our browser.

Connect to a production-ready database ¶

Before we start changing the code of the application remember to tear down the system running

```
$ ./manage.py compose down
Stopping production_web_1   ... done
Stopping production_nginx_1 ... done
Stopping production_db_1    ... done
Removing production_web_1   ... done
Removing production_nginx_1 ... done
Removing production_db_1    ... done
Removing network production_default
```

Thanks to the common interface between repositories, moving from the memory-based `MemRepo` to `PostgresRepo` is very simple. Clearly, as the external database will not contain any data initially, the response of the use case will be empty.

First of all, let's move the application to the Postgres repository. The new version of the endpoint is

application/rest/room.py

```
import os
import json

from flask import Blueprint, request, Response
```

```
from rentomatic.repository.postgresrepo import PostgresRepo
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,
}

postgres_configuration = {
    "POSTGRES_USER": os.environ["POSTGRES_USER"],
    "POSTGRES_PASSWORD": os.environ["POSTGRES_PASSWORD"],
    "POSTGRES_HOSTNAME": os.environ["POSTGRES_HOSTNAME"],
    "POSTGRES_PORT": os.environ["POSTGRES_PORT"],
    "APPLICATION_DB": os.environ["APPLICATION_DB"],
}


@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 = PostgresRepo(postgres_configuration)
    response = room_list_use_case(repo, request_object)

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

As you can see the main change is that `repo = MemRepo(rooms)` becomes `repo = PostgresRepo(postgres_configuration)`. Such a simple change is made possible by the clean architecture and its strict layered approach. The only other notable change is that we replaced the initial data for the memory-based repository with a dictionary containing connection data, which comes from the environment variables set by the management script.

This is enough to make the application connect to the Postgres database that we are running in a container, but as I mentioned we also need to initialise the database. The bare minimum that we need is an empty database with the correct name. Remember that in this particular setup we use for the application a different database (`APPLICATION_DB`) from the one that the Postgres container creates automatically at startup (`POSTGRES_DB`). I added a specific command to the management script to perform this task

manage.py

```
@cli.command()
def init_postgres():
    configure_app(os.getenv("APPLICATION_CONFIG"))
```

```
    try:
        run_sql([f"CREATE DATABASE {os.getenv('APPLICATION_D
    except psycopg2.errors.DuplicateDatabase:
        print(
            (
                f"The database {os.getenv('APPLICATION_DB')}
                "exists and will not be recreated",
            )
        )
```

Now spin up your containers

```
$ ./manage.py compose up -d
Creating network "production_default" with the default drive
Creating volume "production_pgdata" with default driver
Creating production_web_1   ... done
Creating production_nginx_1 ... done
Creating production_db_1    ... done
```

and run the new command that we created

```
$ ./manage.py init-postgres
```

Mind the change between the name of the function `init_postgres` and the name of the command `init-postgres`. You only need to run this command once, but repeated executions will not affect the database.

We can check what this command did connecting to the database. We can do it executing `psql` in the database container

```
$ ./manage.py compose exec db psql -U postgres
psql (13.4 (Debian 13.4-1.pgdg100+1))
Type "help" for help.

postgres=#
```

Please note that we need to specify the user `-U postgres`. That is the user that we created through the variable `POSTGRES_USER` in `config/production.json`. Once logged in, we can use the command `\l` to see the available databases

```
postgres=# \l
                                 List of databases
    Name     |  Owner   | Encoding |  Collate   |   Ctype    
-------------+----------+----------+------------+------------
 application | postgres | UTF8     | en_US.utf8 | en_US.utf8
 postgres    | postgres | UTF8     | en_US.utf8 | en_US.utf8
 template0   | postgres | UTF8     | en_US.utf8 | en_US.utf8
             |          |          |            |            
 template1   | postgres | UTF8     | en_US.utf8 | en_US.utf8
             |          |          |            |            
(4 rows)

postgres=#
```

Please note that the two databases `template0` and `template1` are system databases created by Postgres (see the documentation), `postgres` is the default database created by the Docker container (the name is `postgres` by default, but in this case it comes from the environment variable `POSTGRES_DB` in `config/production.json`) and `application` is the database created by `./manage.py init-postgres` (from `APPLICATION_DB`).

We can connect to a database with the command `\c`

```
postgres=# \c application
You are now connected to database "application" as user "pos
application=#
```

Please note that the prompt changes with the name of the current database. Finally, we can list the available tables with `\dt`

```
application=# \dt
Did not find any relations.
```

As you can see there are no tables yet. This is no surprise as we didn't do anything to make Postres aware of the models that we created. Please remember that everything we are doing here is done in an external system and it is not directly connected with entities.

As you remember, we mapped entities to storage objects, and since we are using Postgres we leveraged SQLAlchemy classes, so now we need to create the database tables that correspond to them.

Migrations

We need a way to create the tables that correspond to the objects that we defined in `rentomatic/repository/postgres_objects.py`. The best strategy, when we use an ORM like SQLAlchemy, is to create and run migrations, and for this we can use Alembic.

If you are still connected with `psql` please exit with `\q`, then edit `requirements/prod.txt` and add `alembic`

```
Flask
SQLAlchemy
psycopg2
pymongo
gunicorn
alembic
```

As usual, remember to run `pip install -r requirements/dev.txt` to update the virtual environment.

Alembic is capable of connecting to the database and run Python scripts (called "migrations") to alter the tables according to the SQLAlchemy models. To do this, however, we need to give Alembic access to the database providing username, password, hostname, and the database name. We also need to give Alembic access to the Python classes that represent the models.

First of all let's initialise Alembic. In the project's main directory (where `manage.py` is stored) run

```
$ alembic init migrations
```

which creates a directory called `migrations` that contains Alembic's configuration files, together with the migrations that will be created in `migrations/versions`. it will also create the file `alembic.ini` which contains the configuration values. The name `migrations` is completely arbitrary, so feel free to use a different one if you prefer.

The specific file we need to adjust to make Alembic aware of our models and our database is `migrations/env.py`. Add the highlighted lines

migrations/env.py

```
import os
```

```
from logging.config import fileConfig

from sqlalchemy import engine_from_config
from sqlalchemy import pool

from alembic import context

# this is the Alembic Config object, which provides
# access to the values within the .ini file in use.
config = context.config

section = config.config_ini_section
config.set_section_option(
    section, "POSTGRES_USER", os.environ.get("POSTGRES_USER"
)
config.set_section_option(
    section, "POSTGRES_PASSWORD", os.environ.get("POSTGRES_P
)
config.set_section_option(
    section, "POSTGRES_HOSTNAME", os.environ.get("POSTGRES_H
)
config.set_section_option(
    section, "APPLICATION_DB", os.environ.get("APPLICATION_D
)

# Interpret the config file for Python logging.
# This line sets up loggers basically.
fileConfig(config.config_file_name)

# add your model's MetaData object here
# for 'autogenerate' support
# from myapp import mymodel
# target_metadata = mymodel.Base.metadata
# target_metadata = None
```

```
from rentomatic.repository.postgres_objects import Base

target_metadata = Base.metadata

# other values from the config, defined by the needs of env.
# can be acquired:
# my_important_option = config.get_main_option("my_important
# ... etc.
```

Through `config.set_section_option` we are adding relevant configuration values to the main Alembic INI file section (`config.config_ini_section`), extracting them from the environment variables. We are also importing the file that contains the SQLAlchemy objects. You can find documentation on this procedure at https://alembic.sqlalchemy.org/en/latest/api/config.html.

Once this is done we need to change the INI file to use the new variables

alembic.ini

```
# the output encoding used when revision files
# are written from script.py.mako
# output_encoding = utf-8

sqlalchemy.url = postgresql://%(POSTGRES_USER)s:%(POSTGRES_P

[post_write_hooks]
# post_write_hooks defines scripts or Python functions that
# on newly generated revision scripts.  See the documentatio
# detail and examples
```

The syntax `%(VARNAME)s` is the basic variable interpolation used by `ConfigParser` (see the documentation).

At this point we can run Alembic to migrate our database. In many cases, you can rely on Alembic's autogeneration functionality to generate the migrations, and this is what we can do to create the initial models. The Alembic command is `revision` with the `--autogenerate` flag, but we need to pass the environment variables on the command line. This is clearly a job for `migrate.py` but let's first run it to see what happens to the database. Later we will create a better setup to avoid passing variables manually

```
$ POSTGRES_USER=postgres\
  POSTGRES_PASSWORD=postgres\
  POSTGRES_HOSTNAME=localhost\
  APPLICATION_DB=application\
  alembic revision --autogenerate -m "Initial"
```

This will generate the file `migrations/versions/4d4c19952a36_initial.py`. Pay attention that the initial hash will be different for you. If you want you can open that file and see how Alembic generates the table and creates the columns.

So far we created the migration but we still need to apply it to the database. Make sure you are running the Docker containers (run `./manage.py compose up -d` otherwise) as Alembic is going to connect to the database, and run

```
$ POSTGRES_USER=postgres\
  POSTGRES_PASSWORD=postgres\
  POSTGRES_HOSTNAME=localhost\
  APPLICATION_DB=application\
  alembic upgrade head
```

At this point we can connect to the database and check the existing tables

```
$ ./manage.py compose exec db psql -U postgres -d applicatio
psql (13.4 (Debian 13.4-1.pgdg100+1))
Type "help" for help.

application=# \dt
              List of relations
 Schema |      Name       | Type  |  Owner
--------+-----------------+-------+----------
 public | alembic_version | table | postgres
 public | room            | table | postgres
(2 rows)

application=#
```

Please note that I used the option `-d` of `psql` to connect directly to the database `application`. As you can see, now we have two tables. The first, `alembic_version` is a simple one that Alembic uses to keep track of the state of the db, while `room` is the one that will contain our `Room` entities.

We can double-check the Alembic version

```
application=# select * from alembic_version;
 version_num
--------------
 4d4c19952a36
(1 row)
```

As I mentioned before, the hash given to the migration will be different in your case, but that value that you see in this table should be consistent with the name of the migration script.

We can also see the structure of the table `room`

```
application=# \d room
                                    Table "public.room"
  Column   |         Type          | Collation | Nullable |
-----------+-----------------------+-----------+----------+-
 id        | integer               |           | not null |
 code      | character varying(36) |           | not null |
 size      | integer               |           |          |
 price     | integer               |           |          |
 longitude | double precision      |           |          |
 latitude  | double precision      |           |          |
Indexes:
    "room_pkey" PRIMARY KEY, btree (id)
```

Clearly, there are still no rows contained in that table

```
application=# select * from room;
 id | code | size | price | longitude | latitude
----+------+------+-------+-----------+----------
(0 rows)
```

And indeed, if you open http://localhost:8080/rooms with your browser you will see a successful response, but no data.

To see some data we need to write something into the database. This is normally done through a form in the web application and a specific endpoint, but for the sake of simplicity in this case we can just add data manually to the database.

```
application=# INSERT INTO room(code, size, price, longitude,
INSERT 0 1
```

You can verify that the table contains the new room with a SELECT

```
application=# SELECT * FROM room;
 id |                 code                 | size | price |
----+--------------------------------------+------+-------+-
  1 | f853578c-fc0f-4e65-81b8-566c5dffa35a |  215 |    39 |
(1 row)
```

and open or refresh http://localhost:8080/rooms with the browser to see the value returned by our use case.

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c08-s02

This chapter concludes the overview of the clean architecture example. Starting from scratch, we created domain models, serializers, use cases, an in-memory storage system, a command-line interface and an HTTP endpoint. We then improved the whole system with a very generic request/response management code, that provides robust support for errors. Last, we implemented two new storage systems, using both a relational and a NoSQL database.

This is by no means a little achievement. Our architecture covers a very small use case, but is robust and fully tested. Whatever error we might find in the way we dealt with data, databases, requests, and so on, can be isolated and tamed much faster than in a system which doesn't have tests. Moreover, the decoupling philosophy not only allows us to provide support for multiple storage systems, but also to quickly implement new access protocols, or new serialisations for our objects.

Last update: 20/08/2021

Section 10 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
- Chapter 4 - Add a web application
- Chapter 5 - Error management
- Chapter 6 - Integration with a real external system - Postgres
- Chapter 7 - Integration with a real external system - MongoDB

Next

- Changelog
- Colophon