Flask microservice with TDD and docker

This is based on the tutorial from tesdriven.io. What it addresses are:

- pipenv for virtual environment and dependencies management
- Flask Restful where resources are build on top of Flask views
- Flask CLI tool to run and manage the app from the command line.
- Debugging in development mode
- Docker for python developer
- Flask-sqlalchemy to support SQLAlchemy in Flask
- Psycopg is the most popular PostgreSQL adapter for the Python Here is a quick summary of things learnt.
- Postgresql docker image
- Pytest for unit and functional testing
- Blueprints for organizing code and components

The folder flask-tdd-docker includes the training code.

Set virtual env

The old way to define virtual environment was to use the following approach:

```
python3.7 -m venv env
source env/bin/activate
deactivate
```

As of today, the approach is to use `pipenv`, where you update the project and development dependencies in a `Pipfile`.

```
pipenv --python 3.7
# start the virtual env
pipenv shell
pipenv install --dev
```

Freeze the dependencies:

```
pipenv lock -r > requirements.txt
```

Define and run the flask app

Define a manage.py to represent the app, and use the Flask CLI shell to manage the app from command line:

```
from flask.cli import FlaskGroup
from project import app


cli = FlaskGroup(app)


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

```
export FLASK_APP=project/__init__.py
# use the Flask CLI from inside the app itself
python manage.py run
```

Run in development mode for debugging.

```
export FLASK_ENV=development
python manage.py run
* Serving Flask app "project/__init__.py" (lazy loading)
* Environment: development
* Debug mode: on
```

With the Flask shell we can explore the data in the application:

```
flask shell
```

Using docker and docker compose

The dockerfile use alpine linux and non root user. The docker compose use volume to mount the code into the container. This is a must for a development environment in order to update the container whenever a change to the source code is made. Then build the image using docker compose.

```
docker-compose build
# then start in detached mode
docker-compose up -d
# Rebuild the docker images
docker-compose up -d --build
# Access app logs
docker-compose logs
# Access to a python shell to control the flask app
docker-compose exec users flask shell
```

Add persistence on Postgresql and use SQLAlchemy

To initialize the postgresql copy a sql file under /docker-entrypoint-initdb.d (creating the directory if necessary).

docker compose section for postgresql:

```
users-db:
    build:
      context: ./project/db
      dockerfile: Dockerfile
    expose:
      - 5432
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
```

Once spun up, Postgres will be available on port 5432 for services running in other containers. Be sure to include dependencies in the app dockerfile

```
# install dependencies
RUN apk update && \
    apk add --virtual build-deps gcc python-dev musl-dev && \
    apk add postgresql-dev && \
    apk add netcat-openbsd && \
    pip install --upgrade pip && \
    pip install --upgrade --user pipenv
```

Also to avoid having the application getting error because it could not contact the database add a entrypoint.sh to loop until the database is accessible before starting the python app.

To access psql use the following docker compose command

```
docker-compose exec users-db psql -U postgres

psql (11.4)
Type "help" for help.

postgres=# \c users_dev
You are now connected to database "users_dev" as user "postgres".
users_dev=# \dt
Did not find any relations.
users_dev=# \q
```

In the `manage.py` file, register a new flask CLI command, `recreate_db`, so that we can run it from the command line like:

```
docker-compose exec users python manage.py recreate_db
```

```
# @cli.command('recreate_db')
def recreate_db():
    db.drop_all()
    db.create_all()
    db.session.commit()
```

Add tests with pytest

While unittest requires test classes, Pytest just requires functions to get up and running.

Define fixtures as reusable elements for future tests

They have a scope associated with them, which indicates how often the fixture is invoked:

- function - once per test function
- class - once per test class
- module - once per test module
- session - once per test session

Some fixture execution guidance

Define python script using 'test_' or '_test.py'. Here is an example of functional testing:

```
def test_ping(test_app):
    client = test_app.test_client()
    resp = client.get('ping')
    data = json.loads(resp.data.decode())
    assert resp.status_code == 200
    assert 'pong' in data['message']
    assert 'success' in data['status']
```

Execute test with pytest: `pytest project/tests/` or with docker compose

```
docker-compose exec users pytest "project/tests"
```

Test coverage

Coverage.py is a popular tool for measuring code coverage in Python-based applications. Now, since we're using Pytest, we'll integrate Coverage.py with Pytest using pytest-cov. In `Pipfile` add `pytest-cov = ">=2.7.1"` then do `pipenv install`

Then once the image is rebuilt, run the following command to assess the test coverage:

```
docker-compose exec users pytest "project/tests" -p no:warnings --
cov="project"
# or using html page
```

```
docker-compose exec users pytest "project/tests" -p no:warnings --
cov="project" --cov-report html
```

Remember: just because you have 100% test coverage doesn't mean you're testing the right things

Code quality

Linting is the process of checking your code for stylistic or programming errors. Although there are a number of commonly used linters for Python, we'll use Flake8 since it combines two other popular linters -- pep8 and pyflakes.

In Pipfile add `flake8 = ">=3.7.8"`, do a `pipenv install` then freeze the dependencies with `pipenv lock -r > requirements.txt`, then rebuild the docker image and run flake8:

```
 docker-compose exec users flake8 project
```

Black helps to format code and apply code formatting:

```
# check
docker-compose exec users black project --check
# see the propose changes
docker-compose exec users black project --diff
# apply the change
docker-compose exec users black project
```

Add Blueprints template

Blueprints are self-contained components, used for encapsulating code, templates, and static files. They are apps within the app. For example REST resource can be defined in Blueprint.

For example to add an api and a resource, define a new py file, and create a blueprint instance:

```
users_blueprint = Blueprint('users', __name__)
api = Api(users_blueprint)
```

Then define a class with functions to support the expected Resource, and add this class to a url to the api.

```
class UsersList(Resource):
    def get(self):
        ...
    def post(self):
        ...
api.add_resource(UsersList, '/users')
```

Finally register the resouce to the flask application:

```
    from project.api.users import users_blueprint
    app.register_blueprint(users_blueprint)
```

See the code in users.py and **init**.py

Factory to create an app needs to be named `create_app`.

Adding admin and model view

Production deployment with gunicorn

Create a specific `Dockerfile.prod` and set the environment variable to run Flask in production mode and use gunicorn as container, and run under a user that is not root.

```
ENV FLASK_ENV production
ENV APP_SETTINGS project.config.ProductionConfig

# add and run as non-root user
RUN adduser -D myuser
USER myuser

# run gunicorn
CMD gunicorn --bind 0.0.0.0:$PORT manage:app
```

Heroku

Using heroku CLI.

```
$ heroku login
# create a app
$ heroku create
Creating app... done, murmuring-shore-37331
https://murmuring-shore-37331.herokuapp.com/ |
https://git.heroku.com/murmuring-shore-37331.git

# login to docker private registry
$ heroku container:login

# create a postgresql with the hobby-dev plan
$ heroku addons:create heroku-postgresql:hobby-dev --app murmuring-shore-37331

Creating heroku-postgresql:hobby-dev on murmuring-shore-37331... free
Database has been created and is available
 ! This database is empty. If upgrading, you can transfer
 ! data from another database with pg:copy
Created postgresql-horizontal-04149 as DATABASE_URL
Use heroku addons:docs heroku-postgresql to view documentation
```

```
# Get database URL
heroku config:get DATABASE_URL --app murmuring-shore-37331
```

The containers used at Heroku are called “dynos.” Dynos are isolated, virtualized Linux containers that are designed to execute code based on a user-specified command.

To build an image for the docker private registry, using the web dyno, that is free.

```
$ docker build -f Dockerfile.prod -t registry.heroku.com/murmuring-shore-
37331/web .
# publish
$ docker push registry.heroku.com/murmuring-shore-37331/web:latest
# test locally
$ docker run --name flask-tdd -e "PORT=8765" -p 5002:8765
registry.heroku.com/murmuring-shore-37331/web:latest

# Release the image, meaning the app will be based on the container image
$ heroku container:release web --app murmuring-shore-37331
Releasing images web to murmuring-shore-37331... done
```

Once the image is "released", the app is accessible via `https://murmuring-shore-37331.herokuapp.com/ping`

Access to logs: `heroku logs --app murmuring-shore-37331`

The users are not yet created, so we can run the CLI `heroku run`:

```
# create DB
heroku run python manage.py recreate_db --app murmuring-shore-37331
# populate the data
heroku run python manage.py seed_db --app murmuring-shore-37331
# Access the database with psql:
# 1. start a local docker postgresql with psql
docker run -ti postgresql bash
> psql postgres://....
> PSQL:
```