# Docker 101 lab

Python Girona - March 2019

Jordi Bagot (https://github.com/jbagot), Xavi Torelló (https://github.com/XaviTorello)

# Agenda

- 1) Brief summary about ***what is a composition***
- 2) Let's go! `//playing together with a composition`
- 3) Play time!
- 4) Doubts, conclusions and shared experiences

# What is a Docker Composition?

## Assumptions

- 1) Everyone has reviewed the Docker101 talk
- 2) Everyone knows what is `Docker` and knows the difference between an `image` and a `container`
- 3) Everyone has been able to experiment with `Docker` and the `dockerhub` ecosystem

# A Composition is

*a.k.a. Docker Compose instance*

- multi-container environment
- with an isolated network
- with automatic NS resolution based on containers name and aliases
- with the ability to inter-relationate and define dependences between each container / service
- with the capability to deploy and integrate public images with specific `build` scripts

# docker-compose is your friend!

# How it looks?

YAML syntax:

```
version: '3'
services:
  web:
    build: .
    ports:
     - "5000:5000"
  redis:
    image: "redis:alpine"
```

---

this will provide two containers

- `web`: that uses local `Dockerfile` definition and binds the `5000/tcp`
- `redis`: that runs an `alpine` tagged `redis` image from DockerHub

# Cheat sheet

### Start composition running containers in background

```
$ docker-compose up -d
```

### Stop composition

```
$ docker-compose down
```

### Build composition

```
$ docker-compose build [$service]
```

```
$ docker-compose start [$service]
```

```
$ docker-compose stop [$service]
```

### Review logs

```
$ docker-compose logs -f [$service]
```

### Rescale service

```
$ docker-compose scale $service=4
```

### Stream container events

```
$ docker-compose events --json
```

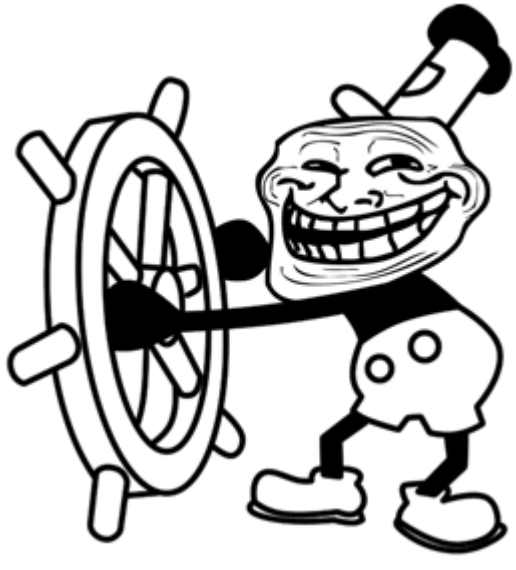
### Drop an interactive shell

```
$ docker-compose -it exec $service bash
```

# Let's go!

Now we're going to extend a real and **very, very very imporant project** with a `Composition` :

https://github.com/pygrn/todos_django (https://github.com/pygrn/todos_django)



This is a `Django` project that serves an example TODOS `API` created by @manelclos (https://github.com/manelclos)

## Prepare the repo

```
$ workon todos_django
```

or `mkproject todos_django` or just `activate` a fresh virtual env

```
$ git clone git@github.com:pygrn/todos_django.git .
```

## Create our build file

- Create a new file named `Dockerfile`

```
FROM python:3.6
ENV PYTHONUNBUFFERED 1
COPY . /code/
WORKDIR /code
RUN pip install -r requirements.txt
```

, that means:

- use `python:3.6` public image, and extend it with
- `export PYTHONUNBUFFERED=1`
- copy all the repo code at `/code`
- assume that current directory will be `/code`
- process and install all project requirements

- Create the image*

```
docker build -t todos_django:latest .
```

, this will tag the resultant image as `todos_django:latest`

- Run the image, just to review what it contains*

```
docker run --rm -it todos_django:latest bash
```

, this will provide an interactive temporal container that uses our image and drops a shell

*: step not necessary, `docker-compose` will build it if needed

## Create our Composition

- Create a new file named `docker-compose.yml`

```yaml
version: '3'
services:
  api:
    build: .
    volumes:
      - .:/code
    ports:
      - "81:8000"
```

, that means:

- hey "mrs compose", this is a version3 composition that should deploy a container serving the `api`
- the image should be builded using our `Dockerfile`
- at run time, `host` project directory should be mounted inside `container` `/code` folder
- `container` `8000/tcp` will be exposed to `host` at `81/tcp`

- Let's going to try to start it

```
$ docker-compose up
```

## WTF??? What's happened?

```
...

Creating network "todos_django_default" with the default driver
Creating todos_django_api_1 ... done
Attaching to todos_django_api_1
todos_django_api_1 exited with code 0

$
```

## We forget to define what it should do!!

```
api:
    ...
    command: ["python", "manage.py", "runserver"]
```

, this tells that `$ python manage.py runserver` should be run once the container is ready!

## , it works, but we should add a DB to our composition!

```
...
api_1  |   File "/usr/local/lib/python3.6/site-packages/django/db/backends/post
gresql/base.py", line 176, in get_new_connection
api_1  |     connection = Database.connect(**conn_params)
api_1  |   File "/usr/local/lib/python3.6/site-packages/psycopg2/__init__.py",
 line 130, in connect
api_1  |     conn = _connect(dsn, connection_factory=connection_factory, **kwas
ync)
api_1  | django.db.utils.OperationalError: could not connect to server: No such
file or directory
api_1  |    Is the server running locally and accepting
api_1  |    connections on Unix domain socket "/var/run/postgresql/.s.PGSQL.543
2"?
api_1  |
```

```yaml
version: '3'
services:

  ....

  db:
    image: kartoza/postgis:latest
    environment:
      - POSTGRES_DB=a_database
      - POSTGRES_USER=a_user
      - POSTGRES_PASS=a_password
      - ALLOW_IP_RANGE=0.0.0.0/0
    ports:
      - 35432:5432
```

, this will

- provide a new service named `db` that will start a PostgreSQL with PostGIS extensions ready
- creating a new database named `a_database`
- granting access for `a_user:a_password`
- allowing connections from any IP
- exposing the container's psql port 5432 as host 35432

# WTF?? Both containers are correctly defined, but the DB is not ready for our web

```
db_1   | 2019-03-05 15:37:23.354 UTC [40] LOG:  database system was shut down a
t 2019-02-01 14:24:17 UTC
db_1   | 2019-03-05 15:37:23.388 UTC [27] LOG:  database system is ready to acc
ept connections
api_1  | Try to load extra settings: settings-production.py
api_1  | Performing system checks...
api_1  |
api_1  | System check identified no issues (0 silenced).
api_1  | Unhandled exception in thread started by <function check_errors.<local
s>.wrapper at 0x7fdc5d447268>
api_1  | Traceback (most recent call last):
api_1  |   File "/usr/local/lib/python3.6/site-packages/django/db/backends/bas
e/base.py", line 213, in ensure_connection
api_1  |     self.connect()
api_1  |   File "/usr/local/lib/python3.6/site-packages/django/db/backends/bas
e/base.py", line 189, in connect
api_1  |     self.connection = self.get_new_connection(conn_params)
api_1  |   File "/usr/local/lib/python3.6/site-packages/django/db/backends/post
gresql/base.py", line 176, in get_new_connection
api_1  |     connection = Database.connect(**conn_params)
api_1  |   File "/usr/local/lib/python3.6/site-packages/psycopg2/__init__.py",
```

## Solution: Use wait scripts!

https://github.com/vishnubob/wait-for-it (https://github.com/vishnubob/wait-for-it)

- Fetch the `wait-for-it.sh` script and save it at `utils/wait-for-it.sh` //ensure that is executable!

```
$ mkdir -p utils && curl https://raw.githubusercontent.com/vishnubob/wai
t-for-it/master/wait-for-it.sh -o utils/wait-for-it.sh && chmod +x util
s/wait-for-it.sh
```

- Prepare an start script! `utils/start-server.sh` //it should be executable!

```
pip install -r requirements.txt
python manage.py migrate
python manage.py runserver 0.0.0.0:8000
```

, this will ensure to review requirements, apply latest pending migrations and start Django!

- Improve our composition to change `web` start command and define a depedency to `db`:

```
api:
    build: .
    volumes:
      - .:/code
    ports:
      - "81:8000"
    command: ["bash", "./utils/wait-for-it.sh", "db:5432", "--", "bash",  "./u
tils/start-server.sh"]
    depends_on:
      - db
```

, this will start our Django once the `5432/tcp@db` is ready to accept connections!

## OK! Now our Django is waiting for the DB, but still breaking!

We should review our `Django` config, **it needs some ENV vars to point to our backend**

```
$ vi todos_project/settings-production.py

DATABASES = {
    'default': {
        # 'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': os.environ.get('DB_NAME'),
        'USER': os.environ.get('DB_USER'),
        'PASSWORD': os.environ.get('DB_PASSWORD'),
        'HOST': os.environ.get('DB_HOST'),
        'PORT': os.environ.get('DB_PORT'),
    },
}
ALLOWED_HOSTS = ['*']
```

## Config the environment

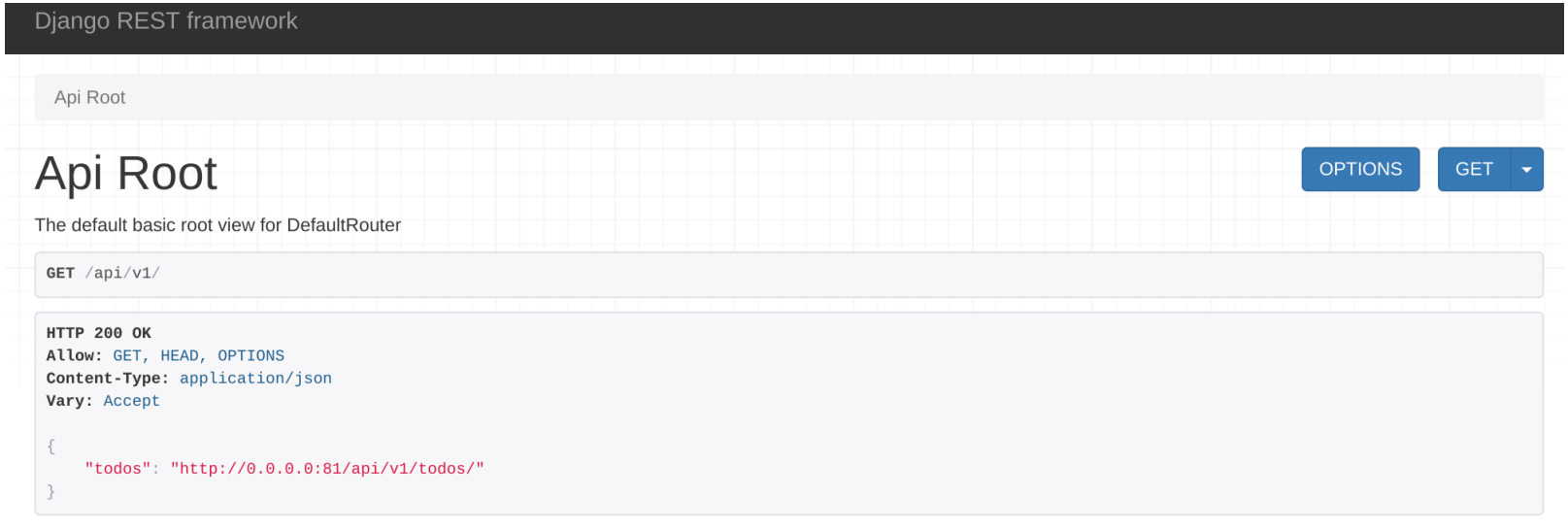
```
...

  api:
    environment:
      - DB_HOST=${DB_HOST}
      - DB_PORT=${DB_PORT}
      - DB_NAME=${DB_NAME}
      - DB_USER=${DB_USER}
      - DB_PASSWORD=${DB_PASSWORD}

...

  db:
    environment:
      - POSTGRES_DB=${DB_NAME}
      - POSTGRES_USER=${DB_USER}
      - POSTGRES_PASS=${DB_PASSWORD}
      - ALLOW_IP_RANGE=0.0.0.0/0
```

- Create an `.env` file

```
DB_HOST=db
DB_PORT=5432
DB_NAME=todos
DB_USER=todos
DB_PASSWORD=this_is_not_a_secure_password
```

# It's magic!! It works!!!

http://0.0.0.0:81/api/v1/ (http://0.0.0.0:81/api/v1/)



## It can be more intense...

We'll try to integrate the `React` frontend created by @francescarpi (http://github.com/francescarpi):

https://github.com/pygrn/todos_react (https://github.com/pygrn/todos_react)

## The idea is to show alternative ways to provide a container as a service

- Create another build script named `Dockerfile_frontend`

```
# Use an alpine-based (ver small base) node image
FROM node:alpine
RUN apk update && apk upgrade && \
    apk add --no-cache bash git openssh

# Prepare our project and their dependencies
RUN mkdir /code
WORKDIR /code
RUN git clone https://github.com/pygrn/todos_react .
RUN sed -i 's;https://server3.microdisseny.com/apps/todos;http://api:8000;g' s
rc/lib/apiclient.js
RUN yarn install

# Directly upload the wait-for-it script to the image
ADD utils/wait-for-it.sh ./
RUN chmod +x wait-for-it.sh
```

- Add the new service!

```
...

  web:
    build:
     context: ./
     dockerfile: Dockerfile_frontend
    command: ["bash", "./wait-for-it.sh", "api:8000", "--", "yarn",  "start"]
    ports:
      - "80:3000"
    depends_on:
      - api
    restart: always
```

## It can be extended with

- build arguments passed from our composition
    - see https://docs.docker.com/compose/compose-file/#build (https://docs.docker.com/compose/compose-file/#build)
    - ie usefull to define which repository tag should be used at build time

- shared volumes
    - https://docs.docker.com/compose/compose-file/#volumes (https://docs.docker.com/compose/compose-file/#volumes)
    - interesting if some kind of low-level sharing should be provided, ie
        - share store between N redis instances
        - map the build output of our nodejs-based app into the access layer (ie nginx)

- custom networks
    - https://docs.docker.com/compose/networking/#specify-custom-networks (https://docs.docker.com/compose/networking/#specify-custom-networks)
    - very useful if some kind of network restrinctions should be applied to our composition, ie
        - isolate a DMZ, a preDMZ and a LAN restricting the exposure and the visibility of our services
        - (re)use already existent networks (powered by other compositions)

- service scalation
    - https://docs.docker.com/compose/reference/scale/ (https://docs.docker.com/compose/reference/scale/)
    - very useful to start N instances of same service, and be able to re-scale if needed, ie for
        - dispatch N workers
        - launch N access layer elements

- ...

## Some interesting utilities

- Clean development environment(s)

- Simple, quick and standardized deployments!
    - previous step to the orchestration

- Local testing
    - Thing about to provide the 3G:
        - Good **migrations**, Good **test data**, Good **tests**! xDD

- Continuous Integration
    - Travis (https://travis-ci.org)
    - CircleCI (https://circleci.com)

# Play time!

- 1) Select one of our favourite projects
- 2) Think which composition can be provided to the project to reach a quick-run / quick-deployable env
- 3) Play with it! If support is needed just warn us!

# It's all!


For more information https://docs.docker.com/compose/ (https://docs.docker.com/compose/)

# Questions?