Roteiro das Práticas

Instalando o python, python-pip e virtualenvwrapper:

- $ sudo pacman -S python python-pip
- $ pip install virtualenvwrapper --user
- $ source $HOME/.local/bin/virtualenvwrapper.sh
- $ mkvirtualenv djangodocker
- OBS: outros comandos:
  - *workon* <*virtualenv*>: carrega (alterna para) este virtualenv
  - *rmvirtualenv* <*virtualenv*>: remove virtualenv
  - *lsvirtualenv*: lista todos os virtualenvs
  - *deactivate*: sai do virtualenv atual
- Quero saber mais sobre o virtualenvwrapper: https://virtualenvwrapper.readthedocs.io/en/latest/
- Quero saber mais sobre o pip: https://pypi.org/project/pip/

Instando o Django e criando um projeto de teste (djangodocker):

- $ pip install django
- $ django-admin startproject djangodocker
- $ cd djangodocker
- Substitua a seção DATABASES default do arquivo *djangodocker/settings.py* por:

```
DATABASES = {
   'default': {
      'ENGINE': 'django.db.backends.mysql',
      'NAME': 'djangodocker_db',
      'USER': 'root',
      'PASSWORD': 'root',
      'HOST': '127.0.0.1',
      'PORT': '3306',
   }
}
```

- Quero saber mais sobre o Django: https://docs.djangoproject.com/en/2.2/intro/tutorial01/

Utilizando um banco de dados mysql via Docker:

- Inicie o container do mysql:
- $ docker run -p 3306:3306 --name mysql -e MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE=djangodocker_db -d mysql
- Explicando os parâmetros:
  - *-p*: especifica a porta externa (127.0.0.1:3306) e interna (3306) do serviço do container
  - *--name*: especifica o nome do container. Se não informado, o docker gera nomes aleatórios
  - *-e*: define o valor de uma variável de ambiente utilizada na imagem
  - *-d*: faz o *detach* do container, liberando o terminal
- Acessando o banco do container:
- $ mysql -h 127.0.0.1 -u root -p djangodocker_db
- Digite a senha ("root") e verifique que o banco iniciou corretamente. Pressione *Ctrl+D* para sair. O banco pode demorar um pouco a subir. Tente novamente, após alguns segundos, caso apareça o erro "*Lost connection to MySQL server at 'handshake: reading inital communication packet*". Acompanhe a inicialização do banco com o comando "*docker logs mysql -f*". Aguarde até que apareça, nos logs, a mensagem "*MySQL init process done. Ready for start up*".
- Quero saber mais sobre sobre a imagem Docker do mysql: https://hub.docker.com/_/mysql
- Quero saber mais sobre docker run: https://docs.docker.com/engine/reference/run/

Executando seu sistema Django:

- Instale o pacote *mysqlclient* do Python (necessário para executar as migrações de banco):
- $ pip install mysqlclient
- Execute as migrações de banco:
- $ python manage.py migrate
- Confira as migrações com o comando:
- $ python manage.py showmigrations
- Edite o arquivo djangodocker/settings.py e adicione a seguinte linha ao final:

STATIC_ROOT = '<*path-to-your-djangodocker-project*>/static'

- Colete os arquivos estáticos com o comando:
- $ python manage.py collectstatic
- Inicie o servidor do Django:
- $ python manage.py runserver 0.0.0.0:8001
- Acesse http://localhost:8001 para verificar se está tudo ok

- Pressione *Ctrl+C* para interromper o servidor

Configurando um proxy reverso com Docker e NGINX:

- Crie um arquivo *nginx.conf* com as seguintes configurações:

```
upstream django_server {
   server <ip-do-localhost-como-visto-pelo-container>:8001 fail_timeout=0;
}
server {
   location / {
      proxy_set_header   Host $host;
      if (!-f $request_filename) {
         proxy_pass http://django_server;
         break;
      }
   }
}
```

- Note que você precisa encontrar o *<ip-do-localhost-como-visto-pelo-container>*. Normalmente este é o IP exibido pelo comando *ifconfig -a* para a interface virtual *docker0*
- Inicie o container do NGINX:
- $ docker run -p 80:80 --name nginx -v <path-to-your-djangodocker-project>/nginx.conf:/etc/nginx/conf.d/default.conf:ro -v <path-to-your-djangodocker-project>/static/:/usr/share/nginx/djangodocker/static/ -d nginx
- Explicando os novos parâmetros:
  - *-v*: cria um *bind mount* (volume externo), fazendo com que o arquivo *nginx.conf* seja disponibilizado, no container, como */etc/nginx/conf.d/default.conf* (arquivo default de configuração do NGINX)
  - :ro: especifica que o *bind mount* é read-only
- Verifique com o comando "*docker ps*" que os dois containers (mysql e NGINX) estão em execução
- Inicie novamente o servidor do Django
- $ python manage.py runserver 0.0.0.0:8001
- Acesse http://localhost para verificar se o proxy reverso está funcionando corretamente
- Pressione *Ctrl+C* para interromper o servidor
- Quero saber mais sobre proxy reverso no NGINX: https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/

Utilizando o Docker Compose:

- Remova os containers em execução:
- $ docker rm -f mysql nginx
- Crie um arquivo *docker-compose.yml* contendo:

```
version: "3.2"
services:
  db:
    image: mysql
    command: --default-authentication-plugin=mysql_native_password
    environment:
      MYSQL_DATABASE: djangodocker_db
      MYSQL_ROOT_PASSWORD: root
    ports:
      - "3306:3306"
  nginx:
    image: nginx
    volumes:
      - ./nginx.conf:/etc/nginx/conf.d/default.conf:ro
      - ./static:/usr/share/nginx/djangodocker/static
    ports:
      - "80:80"
```

- Inicialize todos os serviços com um único comando, executando:
- $ docker-compose up -d
- Aguarde a inicialização do banco e faça uma conexão de teste:
- $ mysql -h 127.0.0.1 -u root -p djangodocker_db
- Execute as migrações:
- $ python manage.py migrate
- Inicie o servidor do Django:
- $ python manage.py runserver 0.0.0.0:8001
- Verifique que o sistema está disponível em http://localhost
- Pressione *Ctrl+C* para interromper o servidor
- Execute "*docker-compose down*" para interromper e remover todos os containers
- Explicando o docker-compose:
    - *docker-compose up*:
        - Se os containers não existem

        - Os containers são criados
    - Caso contrário se as imagens dos containers ou o docker-compose.yml foram atualizadas
        - Os containers são destruídos (mantendo os volumes) e recriados
    - Inicializa os containers na ordem especificada no *depends_on*
  - *docker-compose start*: inicializa containers já criados na ordem especificada no *depends_on*
  - *docker-compose stop*: interrompe containers na ordem inversa da especificada no *depends_on*
  - *docker-compose restart*: interrompe e inicializa os serviços (reinicia na ordem inversa do *depends_on*)
  - *docker-compose down*: interrompe e remove os containers, redes internas e volumes internos
  - *docker-compose pull*: atualiza as imagens utilizadas
- Quero saber mais sobre o Docker Compose: https://docs.docker.com/compose/

Problemas do setup anterior:

- Um ambiente Python com suas dependências ainda precisa ser instalado na máquina do desenvolvedor. Isso pode não ser trivial caso:
  - Necessite-se de versões específicas das dependências e de outros serviços;
  - Os desenvolvedores utilizem diferentes distribuições Linux com disponibilidade diferentes destas versões específicas de dependências.
- Se vários desenvolvedores trabalham no mesmo projeto, cada um deles precisaria ter um arquivo *nginx.conf* diferente (por conta da IP do servidor Django). Consequentemente, este arquivo não poderia ser versionado.

Vamos ver como resolver estes problemas.

Movendo a aplicação para um container Docker:

- Vamos criar uma nova imagem Docker contendo a nossa aplicação Django. Crie um arquivo chamado *Dockerfile* contendo:

```
FROM python

WORKDIR /app
COPY djangodocker djangodocker
COPY manage.py requirements.txt /app/
RUN pip install -r requirements.txt && \
   python manage.py collectstatic --noinput

EXPOSE 8001
```

CMD ["python", "manage.py", "runserver", "0.0.0.0:8001"]

- Crie o arquivo requirements.txt contendo todas as dependências utilizadas pela aplicação:

Django==2.2.4
mysqlclient==1.4.3

- Teste a criação da imagem com o comando:
- $ docker build -t django_image:v1 .
- Verifique que a imagem foi criada com sucesso com o comando "*docker images"*
- Vamos agora integrar esta nova imagem no Docker Compose. Altere o arquivo *docker-compose.yml* para conter a seguinte configuração:

```
version: "3.2"
services:
  app:
    build: .
    depends_on:
      - db
  db:
    image: mysql
    command: --default-authentication-plugin=mysql_native_password
    environment:
      MYSQL_DATABASE: djangodocker_db
      MYSQL_ROOT_PASSWORD: root
    ports:
      - "3306:3306"
  nginx:
    image: nginx
    volumes:
      - ./nginx.conf:/etc/nginx/conf.d/default.conf:ro
      - ./static:/usr/share/nginx/djangodocker/static
    ports:
      - "80:80"
    depends_on:
      -app
```

- Este arquivo define três serviços: *app*, *mysql* e *nginx*. Quando os containers são criados, o Docker Compose coloca estes três containers na mesma "rede virtual". Com isso, um serviço pode acessar o IP de outro serviço simplesmente utilizando o seu nome.
- Com isso, vamos modificar as configurações de banco de dados da nossa aplicação. Altere a seção DATABASES do arquivo *djangodocker/settings.py* para o seguinte:

```
DATABASES = {
   'default': {
      'ENGINE': 'django.db.backends.mysql',
      'NAME': 'djangodocker_db',
      'USER': 'root',
      'PASSWORD': 'root',
      'HOST': 'db',          # Acessando o banco atraves do nome do servico
      'PORT': '3306',
   }
}
```

- Vamos também alterar a configuração do NGINX para encontrar o IP do servidor Django através do nome do serviço. Altere o arquivo *nginx.conf* para:

```
upstream django_server {
   server app:8001 fail_timeout=0;  # Acessando o servidor Django atraves do nome do servico
}
```

- Inicialize os três containers:
- $ docker-compose up -d
- Se você quiser recriar a imagem antes de inicializar os containers, execute:
- $ docker-compose up –build -d
- Aguarde a inicialização do banco e execute as migrações (desta vez, precisamos executar a migração de dentro do container):
- $ docker-compose exec app python manage.py migrate
- Se você tentar acessar a aplicação em http://localhost você verá que irá aparecer uma mensagem "*502 bad gateway*". Isso ocorre porque há uma dependência cíclica entre os serviços *app* e *nginx*. Vamos reinicializar o container da aplicação para que o NGINX funcione corretamente:
- $ docker-compose restart app
- Acesse a aplicação em http://localhost e veja que o sistema passa a funcionar

- Execute "*docker-compose down*" para interromper e remover todos os containers

Utilizando a imagem nginx-proxy:

- A imagem nginx-proxy fornece uma forma fácil de configuração de proxies reversos com a criação automática do arquivo de configuração nginx.conf
- Para isso, vamos alterar o arquivo docker-compose.yml para o seguinte:

```
version: "3.2"
services:
  app:
    build: .
    environment:
      VIRTUAL_HOST: app1.ifba.edu.br
    depends_on:
      - nginx-proxy
  db:
    image: mysql
    command: --default-authentication-plugin=mysql_native_password
    environment:
      MYSQL_DATABASE: djangodocker_db
      MYSQL_ROOT_PASSWORD: root
    ports:
      - "3306:3306"
  nginx-proxy:
    image: jwilder/nginx-proxy
    depends_on:
      - "db"
    ports:
      - "80:80"
    volumes:
      - /etc/nginx/vhost.d
      - /usr/share/nginx/html
      - /var/run/docker.sock:/tmp/docker.sock:ro
```

- A imagem nginx-proxy procura por serviços contendo a variável de ambiente VIRTUAL_HOST (em negrito acima) e cria, automaticamente, configurações de proxy reverso para estes serviços, utilizando o nome informado (neste caso: *app1.ifba.edu.br*) \o/

- Agora vamos simular a existência do nome *app1.ifba.edu.br* alterando o arquivo */etc/hosts*. Execute "*sudo /etc/hosts*" e adicione a seguinte linha:

```
127.0.0.1       app1.ifba.edu.br
```

- Altere o arquivo djangodocker/settings.py para incluir este nome na lista de hosts permitidos. Altere a linha ALLOWED_HOSTS para:

```
ALLOWED_HOSTS = [ "app1.ifba.edu.br" ]
```

- Reinicie os serviços, recriando a imagem da nossa aplicação:
- $ docker-compose up --build -d
- Aguarde a inicialização do banco e execute as migrações:
- $ docker-compose exec app python manage.py migrate
- Se você tentar acessar a aplicação em http://app1.ifba.edu.br você verá que irá aparecer uma mensagem "*502 bad gateway*". Isso ocorre porque o banco demora a inicializar na sua primeira execução (o banco e configuções default precisam ser criados) e os outros serviços não estão aguardando a finalização desta inicialização. Vamos interromper e reinicializar os serviços:
- $ docker-compose stop
- $ docker-compose start
- Note que agora o sistema funciona perfeitamente
- NOTA: você pode criar um usuário no Django para testar a aplicação admin executando:
- $ docker-compose exec app python manage.py createsuperuser

Aguardando a inicialização do banco:

- Na versão 2 do Docker Compose pode-se utilizar *conditions* nas cláusulas *depends_on*:

```
version: "2.1"
...
  db:
    ...
    healthcheck:
      test: [ "CMD", "mysqladmin" ,"ping", "-h", "localhost" ]
      timeout: 5s
      retries: 10
  nginx-proxy:
    image: jwilder/nginx-proxy
```

```
    depends_on:
      db:
        condition: service_healthy
...
```

- Na versão 3 do Docker Compose, é necessário criar um *script* que verifica se o serviço já foi inicializado e utilizar este script na cláusula *command* do arquivo docker-compose.yml. Geralmente utilize-se https://github.com/vishnubob/wait-for-it para isso.

Movendo os dados do banco para um volume externo:

- Os container do banco mysql usa um volume *unnamed* interno para armazenar os dados. Isto significa que os dados são destruídos sempre que o container é destruído e recriado. Podemos manter os dados (migrações) configurando o serviço mysql para usar um volume externo (*mount bind*). Isto é importante porque containers são destruídos e recriados sempre que uma nova versão da imagem é obtida. Para configurar o volume externo, altere o serviço do mysql no arquivo docker-compose.yml para o seguinte:

```
  db:
    image: mysql
    command: --default-authentication-plugin=mysql_native_password
    volumes:
      - mysql:/var/lib/mysql
    environment:
      MYSQL_DATABASE: djangodocker_db
      MYSQL_ROOT_PASSWORD: root
    ports:
      - "3306:3306"
    healthcheck:
      test: ["CMD-SHELL", 'mysqladmin ping']
      interval: 5s
      timeout: 2s
      retries: 20
volumes:
  mysql:
```

- Inicialize os serviços com "*docker-compose up -d*" e execute as migrações. Após isso, destrua todos os containers com "*docker-compose down*" e inicialize os serviços novamente. Perceba que as migrações foram preservadas.
- Volumes externos são criados, no arquivo docker-compose.yml, de duas formas:
  - Volumes *unnamed*: "*- /var/lib/mysql*"
  - Volumes *named*:

```
version: "3.2"
services:
  ...
  db:
    image: mysql
    command: --default-authentication-plugin=mysql_native_password
    volumes:
      - mysql:/var/lib/mysql
    environment:
      MYSQL_DATABASE: djangodocker_db
      MYSQL_ROOT_PASSWORD: root
    ports:
      - "3306:3306"
    healthcheck:
      test: ["CMD-SHELL", 'mysqladmin ping']
      interval: 5s
      timeout: 2s
      retries: 20
  ...

volumes:
  mysql:
```

- Note como eles são apresentados de forma diferente no resultado do comando "*docker volume ls*"

Suportando diferentes configurações para os ambientes de desenvolvimento, testes/CI e produção:

- Vamos criar diferentes arquivos *requirements.txt* para indicar as dependências a serem instaladas em cada tipo de ambiente. Para isso, vamos criar uma pasta *requirements*:
- $ mkdir requirements
- Agora, vamos criar os arquivos de dependência para cada ambiente:

- $ mv requirements.txt requirements/base.txt
- Crie o arquivo requirements/development.txt contendo:

```
-r base.txt
pytest==5.0.1
pytest-django==3.5.1
```

- Crie o arquivo requirements/production.txt contendo:

```
-r base.txt
```

- Agora, vamos alterar o arquivo Dockerfile para receber, via argumento, qual arquivo de dependências deverá ser utilizado na construção da imagem:

```
FROM python

ARG requirements=requirements/production.txt
WORKDIR /app

COPY djangodocker djangodocker
COPY manage.py /app/
COPY requirements/ /app/requirements/

RUN pip install -r $requirements && \
   python manage.py collectstatic --noinput

EXPOSE 8001

CMD ["python", "manage.py", "runserver", "0.0.0.0:8001"]
```

- Podemos informar o valor do argumento *requirements* no arquivo docker-compose.yml:

```
version: "3.2"
services:
   app:
      build:
         context: .
```

```
      args:
        requirements: requirements/development.txt
    environment:
      VIRTUAL_HOST: app1.ifba.edu.br
    depends_on:
      - nginx-proxy
```

- Precisamos também utilizar diferentes arquivos *settings.py* da nossa aplicação Django. Utilizaremos uma abordagem semelhante. Execute os seguintes comandos:
- $ mkdir djangodocker/settings
- $ mv djangodocker/settings.py djangodocker/settings/production.py
- $ cp djangodocker/settings/production.py djangodocker/settings/development.py
- Edite o arquivo djangodocker/settings/production.py para desabilitar o debug:

```
...
DEBUG = False
...
```

- Finalmente, precisamos indicar ao container Docker qual arquivo de *settings* utilizar. Diferente do arquivo de dependências, este valor deve estar disponível em *run-time* (usando ENV) e não em *build-time* (usando ARG):

```
FROM python

ARG requirements=requirements/production.txt
ENV DJANGO_SETTINGS_MODULE=djangodocker.settings.production
...
```

- No arquivo docker-compose.yml, podemos indicar que queremos usar o settings de desenvolvimento:

```
version: "3.2"
services:
  app:
    build:
      context: .
      args:
```

```
        requirements: requirements/development.txt
    environment:
      DJANGO_SETTINGS_MODULE: djangodocker.settings.development
      VIRTUAL_HOST: app1.ifba.edu.br
    depends_on:
- nginx-proxy
...
```

Executando testes:

- Define um novo arquivo de requirements chamado testing.txt contendo:

```
-r base.txt
pytest==5.0.1
pytest-django==3.5.1
```

- Faça com que development.txt herde de testing.txt em vez de base.txt
- Crie um novo arquivo de settings chamado djangodocker/settings/testing.py, contendo:

```
from .development import *

DATABASES = {
  'default': {
    'ENGINE': 'django.db.backends.sqlite3',
    'NAME': ':memory:',
  }
}
```

- Crie o arquivo de configuração do PyTest (pytest.ini), contendo:

```
[pytest]
DJANGO_SETTINGS_MODULE = djangodocker.settings.testing

python_files = tests.py test_*.py *_tests.py
```

- Vamos escrever o nosso primeiro teste:
- $ python manage.py startapp simpleapp
- $ mv simpleapp djangodocker/

- Insira esta nova aplicação no arquivo djangodocker/settings/development.py:

```
INSTALLED_APPS = [
   'django.contrib.admin',
   'django.contrib.auth',
   'django.contrib.contenttypes',
   'django.contrib.sessions',
   'django.contrib.messages',
   'django.contrib.staticfiles',
   'djangodocker.simpleapp',
]
```

- Vamos criar um teste simples. Edite o arquivo djangodocker/simpleapp/tests.py para que ele contenha:

```
from django.test import TestCase

# Create your tests here.a

class TestSimpleApp:
   def test_one(self):
      x = "my simple app test"
      assert 'simple app' in x
```

- Execute o teste:
- $ py.test
- Executando o teste no container:
- $ docker-compose run -e DJANGO_SETTINGS_MODULE=djangodocker.settings.testing --no-deps --rm app py.test

Integração Contínua com o Travis

- Acesse https://travis-ci.org/ e faça o login com sua conta do GitHub
- Na página do seu profile, você verá seus projetos do GitHub
- Ao ativar o projeto no Travis, ele fica esperando por um git push e executa as ações presentes no arquivo *.travis.yml* do repositório, contendo:

```
sudo: required
```

```
dist: trusty

services:
- docker

install:
- docker-compose pull
- docker-compose build

script:
- docker-compose run -e DJANGO_SETTINGS_MODULE=djangodocker.settings.testing --no-deps --rm app py.test
```

- Faça um commit com este novo arquivo no seu repositório e veja o Travis executando a Integração Contínua.

Publicando imagens no Docker Hub:

```
docker build . -f myuser/myimage:latest
docker login
docker push myuser/myimage:latest
```

Publicando através da Integração Contínua:

- Crie um arquivo scripts/deploy.sh, contendo:

```
#!/usr/bin/env bash

set -e;

DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
cd "$DIR/.."

image_tag="latest";
image_full_name="sandroandrade/djangodocker:$image_tag";

echo "Building image '$image_full_name'";
```

```
docker build . -t "$image_full_name";

echo "Authenticating";
echo "$DOCKER_PASS" | docker login -u="$DOCKER_USERNAME" --password-stdin;

echo "Pushing image '$image_full_name'";
docker push "$image_full_name";
echo "Push finished!";

exit 0;
```

- Configure as variáveis DOCKER_USERNAME e DOCKER_PASS no Travis.
- Configure um passo de deploy no arquivo .travis.yml:

```
sudo: required
dist: trusty

services:
- docker

install:
- docker-compose pull
- docker-compose build

script:
- docker-compose run -e DJANGO_SETTINGS_MODULE=djangodocker.settings.testing --no-deps --rm
app py.test

deploy:
  - provider: script
    script: bash scripts/deploy.sh
    on:
      branch: master
    skip_cleanup: true
```