Docker Virtualization in practice

In the following, we will see how the popular Docker virtualization/containers can be used in practice.

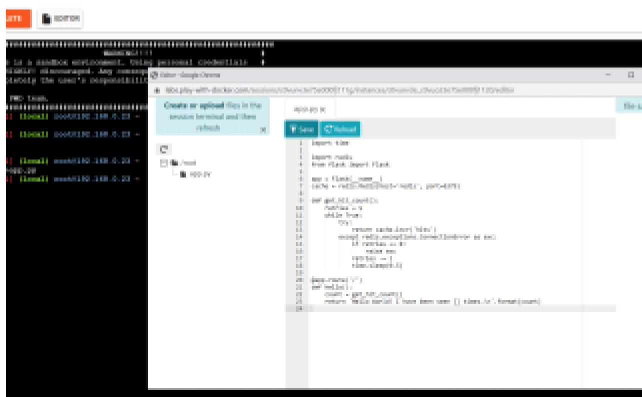
Please log in at http://docker.iit.uni-miskolc.hu/ . Then click the green start button.

After pressing the "+ Add new instance" button on the left side, we will see the following screen:

Create a file called *app.py* by running the following command and press Ctrl+d to finish:

```
cat>app.py
```

By pressing the Editor button next to the delete button, a 'file editor' will appear in a new window. Open the just created *app.py*.



Copy the following Python code into the editor:

```
import time

import redis
from flask import Flask

app = Flask(__name__)
cache = redis.Redis(host='redis', port=6379)

def get_hit_count():
    retries = 5
    while True:
```

```
        try:
            return cache.incr('hits')
        except redis.exceptions.ConnectionError as exc:
            if retries == 0:
                raise exc
            retries -= 1
            time.sleep(0.5)

@app.route('/')
def hello():
    count = get_hit_count()
    return 'Hello World! I have been seen {} times.\n'.format(count)
```

This Python program demonstrates one of the simplest web server/frameworks called Flask. The code snippet above implements a counter that counts visitors using the cache system called **redis**.

Let's create a file called **requirements.txt** in the same way, as before:

```
cat>requirements.txt
```

Then, using the editor, copy the following lines:

```
flask
redis
```

This lines define the dependencies of our application. In our case, these are the *flask* framework and the *redis* cache. This is because the virtual machine will start with an empty configuration and we will use the *requirements.txt* file to install the dependencies. That is, we do not install anything manually, only in a standard way.

Creating a **Dockerfile**

Let's create a file called **Dockerfile** with the following content as usual:

```
FROM python:3.7-alpine
WORKDIR /code
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
RUN apk add --no-cache gcc musl-dev linux-headers
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
EXPOSE 5000
COPY . .
CMD ["flask", "run"]
```

We need to stop here for a little explanation. The Dockerfile's job is to define the step-by-step process of creating a virtual machine.

We define the following line by line:

- Create a starting virtual machine (image) with python 3.7 support and a linux kernel called alpine.
- our working directory will be /code.
- Let's set two environment variables that flask needs to serve.
- Install gcc and other dependencies. (this is necessary because Python compiles many packages from c/c++ source files)
- copy requirements.txt into the working directory (this is necessary because the virtual machine has its own file system, it cannot read the files next to the Dockerfile directly.)
- EXPOSE command opens a tcp port to the outside (in this case, 5000)
- copy everything to the working directory

the last line defines the startup command after installation, in this case: “flask run”

Soronként a következőket definiáljuk:

- Hozzon létre egy kiinduló virtuális gépet (image) a python 3.7-es támogatással és a **alpine** nevű linux kernellel.
- a munkakönyvtárunk a /code lesz.
- Állítsunk be két környezeti változót, ami a flask-nak szükséges a kiszolgáláshoz.
- Telepítsük a gcc-t és más függőségeket. (ez azért kell, mert a Python sok csomagot c/c++ forrás állományokból fordít)
- másoljuk be a requirements.txt-t a munkakönyvtárba (ez azért kell, mert a virtuális gépnek saját fájlrendszere van, a Dockerfile mellett lévő állományokat nem tudja közvetlenül olvasni.)
- EXPOSE parancs tcp portot nyit meg kifelé (jelen esetben az 5000-est)
- mindent másoljuk be a munkakönyvtárba
- az utolsó sor a telepítés utáni indító parancsot definiálja, jelen esetben: “flask run”

Compose állomány létrehozása

Más leírásokban ennél a pontnál elindítják a virtuális gépet. Mi nem tesszük meg, hanem továbblépünk a **docker-compose** lehetőségeire, amivel rugalmasan tudunk több virtuális gépet egyszerre kezelni. Nem feltétlenül kell Dockerfile-t sem létrehozni, ha Interneten is elérhető szabványos konfigurációkat használunk.

Hozzuk létre a docker-compose.yml állományt a szokásos módon:

```
version: "3.3"
services:
  web:
    build: .
    ports:
      - "80:5000"
  redis:
    image: "redis:alpine"
```

Ez az állomány szolgáltatásokban gondolkodik. Minden **service** egy különálló docker image, viszont a nevükre hivatkozva belsőleg elérik egymást. A fenti konfiguráció **web** elnevezésű szolgáltatását a **build: .** miatt a Dockerfile alapján hozzuk létre, viszont a másik **redis** szolgáltatást a szabványos “redis:alpine” konfiguráció alapján használjuk.

A **web** esetén a belsőleg kinyitott 5000-es portot láthatóvá tesszük a 80-as port-on.

Indítsuk el a következő parancsot és várjuk meg ameddig lefut:

```
docker-compose up
```

A következő képernyőhöz hasonlót kell látnunk, ha mindent jól állítottunk be előzőleg. Nyomjuk meg a nyíllal jelölt gombot.

Fejlesztési lehetőségek

Frissítsük többször a böngészőt, láthatjuk, hogy a látogatásszám dinamikusan frissül.

Ctrl + c segítségével megállíthatjuk a futtatást.

cseréljük le a docker-compose.yml tartalmát a következőre:

```
version: "3.3"
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
    environment:
      FLASK_ENV: development
  redis:
    image: "redis:alpine"
```

A **volumes** beállítja, hogy a virtuális gépben belül levő könyvtár, jelen esetben a working directory a gazda rendszerhez legyen kötve, illetve kimásolva. Ha nem a gyökérbe szeretnénk mappelni, akkor pl: ./mycode:/code is megadható, de a mycode könyvtárnak az indítás előtt léteznie kell.

Indítsuk el újra a rendszert:

```
docker-compose up
```

Majd látható, hogy a konzolban developer módra kapcsoltunk. Módosítsuk az editor segítségével a

app.py-t, mondjuk az utolsó függvényben a kiírás szövegét és frissítsük a böngészőt.

Docker compose parancsok

Futó virtuális gépek listázása:

```
docker-compose ps
```

Egy adott instance milyen környezeti változókat használ?

```
docker-compose run web env
```

Hogyan állíthatjuk le a szolgáltatásokat?

```
docker-compose stop
```

Hogyan tudunk teljesen letörölni mindent leállítás után?

```
docker-compose down --volumes
```

Hogyan tudunk shellbe belépni egy konténeren belül?

```
docker-compose exec <containername> sh
```

Honnan tudok előre elkészített minta container-eket letölteni?

https://github.com/docker/awesome-compose

From:
https://edu.iit.uni-miskolc.hu/ - **Institute of Information Science - University of Miskolc**

Permanent link:
https://edu.iit.uni-miskolc.hu/tanszek:oktatas:iss_t:docker?rev=1680454606

Last update: **2023/04/02 16:56**