Docker Virtualization in practice

We will see how the popular Docker virtualization/containers can be used in practice.

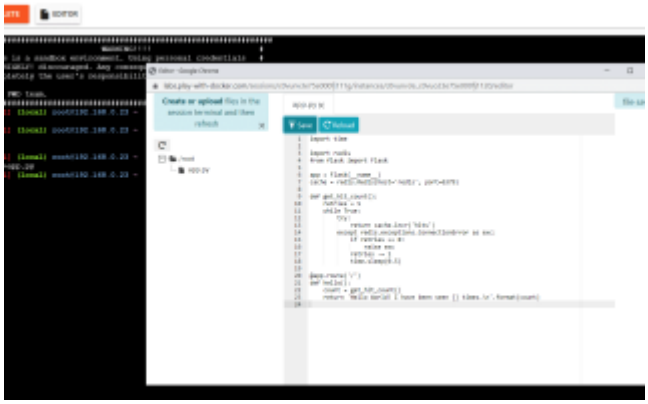
Please log in at http://docker.iit.uni-miskolc.hu/ . Then click the green start button. (if does not work please use the official version: https://labs.play-with-docker.com/)

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”

Creating a docker-compose file to manage lifecycle

In other tutorials, the virtual machine is started at this point. We don't do it, but move on to the possibilities of **docker-compose.yml**, which allows us to flexibly manage several virtual machines at the same time. It is not necessary to create a Dockerfile either, if we use standard configurations available on the Internet.

Create the docker-compose.yml file as usual:

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

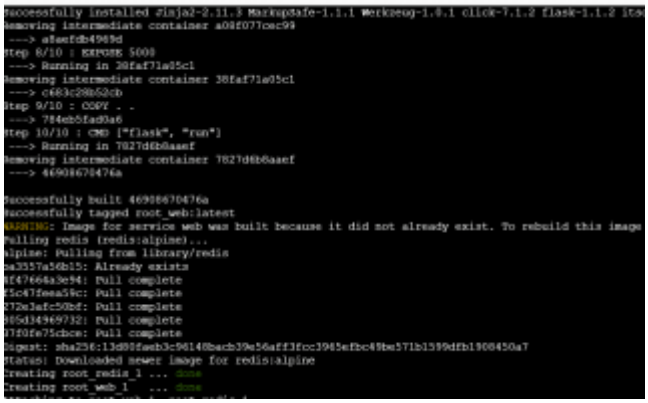
Each *service* is a separate docker image, but they reach each other internally by referring to their names. The service named *web* in the above configuration is provided by *build: .* because of this, we create it based on the *Dockerfile*, but we use the other redis service based on the standard “redis:alpine” configuration.

In the case of the *web*, the internally opened port 5000 is made visible on port 80.

Let's start the following command and wait until it runs:

```
docker-compose up
```

We should see something similar to the next screen, if everything was set up correctly previously.



Possibilities for development

Refresh the browser several times, we can see that the number of visits is updated dynamically.

Ctrl + c can be used to stop the execution.

replace the contents of *docker-compose.yml* with the following:

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

volumes sets the directory inside the virtual machine, in this case the working directory ./ is connected to the host system. (it means, all the file changes will shared between the host and container.) If you don't want to map to the root, you can enter e.g.: ./mycode:/code, but the **mycode** directory must exist before starting.

Let's restart the system:

```
docker-compose up
```

Then you can see that we switched to developer mode in the console. Use the editor to modify *app.py*, say the text of the statement in the last function, and update the browser.

Docker compose commands

List running virtual machines::

```
docker-compose ps
```

What environment variables does a given instance use?

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

How can we stop the services?

```
docker-compose stop <service-name>
```

How can we completely wipe everything after shutdown?

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

How can we enter a shell inside a container?

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

How can I see the logs?

```
docker-compose logs <containername>
```

Where can I download pre-made sample containers?

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

Source code can be found here: https://github.com/knehez/isi in folder example_1.