ulm university universität
uulm

Tools & Concepts for (Cloud) Deployments

Exercise 5: Containers

Christopher B. Hauser

Institute of Information Resource Management, Ulm University

2018-06-07

Exercise 5: Containers

Welcome to exercise 5. This time we will have the following lessons:

1. Concept of Containers
2. Mediawiki with Docker

Lessons learned

- Learn about Containers and their relation to virtual machines
- How to install and use LXC containers
- How to install and use Docker containers
- Multi-tier applications like Mediawiki with Docker and Dockerfile
- Docker-compose for single-host orchestration

Lesson 1: Concept of Containers

Our Cloud Stack so far had three layers: Cloud Platform, Virtual Resource, and Application Component. This means, that the applications are installed directly on top of the virtual operating system. In this lesson we will introduce a fourth layer: the application components are no longer installed on the operating system but are deployed inside containers.

Cloud Stack	Example	Deployment Tool
Application Component	Mediawiki	(Bash) scripts
Containers	Docker	Dockerfile
Virtual Resource	Instance m1.small	Terraform
Cloud Platform	OpenStack	-

Research: What are containers and why to use them?

Before we start to work with containers, we need to understand why containers exist, and what their benefits are. This blog post [1] gives a nice introduction. The most popular container software is Docker. On the Docker page, they compare containers and virtual machines and how they fit together [2].

In the practical part of this exercise we will use Docker [3] and LXC [4] as container software. Become familiar with both tools, to understand their similarities and differences.

[1] https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-an-overview-of-containerization

[2] https://www.docker.com/what-container

[3] https://www.docker.com/

[4] https://linuxcontainers.org/

Question: Containers, LXC and Docker

- What architectural implications are required for an application to run in containers?
- What are pros/cons of containers compared to virtual machines?

Task: Create Containers with LXC

Clean up your openstack workspace: remove unnecessary virtual machines, release floating IPs, remove unused snapshots.

Create a new virtual machine named “lxc” in openstack with flavor small from Ubuntu 16.04, attach a floating IP and log into this VM via SSH. Inside the VM, let’s install software for LXC. We will follow the official *lxc getting started guide* [1]

```
sudo apt-get update
sudo apt-get install -y lxc

# some configurations needed
echo ”ubuntu veth lxcbr0 10” | sudo tee -a /etc/lxc/lxc-usernet
mkdir -p ~/.config/lxc
cp /etc/lxc/default.conf ~/.config/lxc

echo ”lxc.id_map = u 0 $(cat /etc/subuid | grep ubuntu | cut -d ':' -
f 2,3 | tr ':' ' ')
lxc.id_map = g 0 $(cat /etc/subgid | grep ubuntu | cut -d ':' -
f 2,3 | tr ':' ' ')” \
    >> ~/.config/lxc/default.conf

# reboot the vm
sudo reboot
```

Lets create and start an LXC container according to [1]:

```
# create a ubuntu container
lxc-create -t download -n my-container
# when asked, select Distribution: ubuntu, Release: xenial, Architecture: i386

# validate that the container exists and is STOPPED
lxc-ls -f

# start the container
lxc-start -n my-container -d

# validate that the container is now RUNNING
lxc-ls -f

# Count the number of processes in the VM
ps -ef | wc -l

# log into the container
```

```
lxc-attach -n my-container

## now, let's explore the inside of the container...

# Count the number of processes in the container
ps -ef | wc -l

# Check the networking inside the container
ip addr show

# exit container
exit

# stop and destroy container
lxc-stop --name my-container
lxc-destroy --name my-container

## now back on the host run
## "ps -ef | wc -l" and "ip addr show"
## to see the differences
```

The newest version of lxc provides besides the lxc-{create,start,ls,stop,destroy,...} commands also the `lxc` command [2]:

```
# browse images
lxc image list images

# launch a container
lxc launch ubuntu:16.04 web1

# limit container to 1 cpu core
lxc config set web1 limits.cpu 1

# log in and install a web server
lxc exec web1 bash
apt-get install -y apache2
exit

# clone the container to web2
```

```
lxc copy web1 web2
lxc start web2

## note: you just scaled horizontally - in a few seconds! :-)

# list the containers
lxc list
```

As you can see, lxc containers can be cloned very quickly, which allows a fast and flexible scale out of application components. The lxc commands can be called by a script to automate the deployment and management of lxc containers.

[1] https://linuxcontainers.org/lxc/getting-started/

[2] https://www.jamescoyle.net/cheat-sheets/2540-lxc-2-x-lxd-cheat-sheet

Question: LXC Containers

- Where are more processes running: on the vm operating system or inside the container? And why?
- How does the networking look like? Why is it a good idea to have a private IP for each containers?
- What resources can be limited via `lxc config set [container] limits.*` command?

Task: Create Containers with Docker

Create a new virtual machine named "docker" in openstack with flavor small from Ubuntu 16.04, attach a floating IP and log into this VM via SSH. Inside the VM, let's install software for Docker [1].

```
sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg \
    | sudo apt-key add -
sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
sudo apt-get update
sudo apt-get install -y docker-ce
```

```
sudo usermod -aG docker ubuntu
sudo systemctl enable docker
sudo systemctl start docker
```

You can validate that docker is installed by checking the version `docker --version` (should be something like 18.03.1-ce). *Before we continue, log out and log in again via ssh - do activate the new group from the usermod command.* Next, let's create a container and explore the docker commands ...

```
# create and start a container
docker run -d --name web1 ubuntu:16.04 sleep infinity

# list containers
docker ps -a

# list images
docker images

# attach to container and install webserver
docker exec -ti web1 bash
apt-get update
apt-get install -y apache2
exit

# Store container as an image
docker commit web1 myweb:v1

# list images
docker images

# Start another container from this image
docker run -d --name web2 myweb:v1

# do some more experiments ...
# can you manage to access the web servers?
# hint: docker run has a port forwarding parameter

# stop and remove container
docker stop web1 web2
```

```
docker rm web1 web2
```

Besides installing containers manually, Docker introduces the `Dockerfile` - a build description for docker containers. Let's try to build an example from a Dockerfile.

Create a folder `docker-test` in the home directory of your docker vm. Create and open a file `Dockerfile` inside the `docker-test` folder and place the following content:

```
FROM ubuntu:16.04
RUN apt-get update; apt-get install -y apache2
RUN mkdir /opt/init
RUN echo '#!/bin/bash \nset -x \n/usr/sbin/apache2ctl -DFOREGROUND' > /opt/init/ent
RUN chmod +x /opt/init/entrypoint
ENTRYPOINT "/opt/init/entrypoint"
```

Next, let's build an image from this Dockerfile and start a new container. Inside the folder `docker-test`:

```
# Build the image
docker build -t myreg/web .

# list images
docker images

# start container
docker run -d --name web1 myreg/web
```

We created two images myweb and myreg/web using two different approaches:

```
REPOSITORY        TAG             IMAGE ID         CREATED           SIZE
myreg/web         latest          8095a8698075     2 seconds ago     253MB
myweb             v1              fb191309fa9f     About a minute ago  253MB
ubuntu            16.04           5e8b97a2a082     7 days ago        114MB
```

Docker Hub [2] is a central registry for Docker images. Tons of per-packaged software is available already, like database servers, web servers, ...

[1] https://docs.docker.com/engine/installation/linux/ubuntu/#install-using-the-repository

[2] https://hub.docker.com/

Question: Docker and Docker Hub

Now that you are familiar with the basics of Docker:

- What are the differences between a Dockerfile and a Docker image? Can you imagine pros/cons?

- How does a typical workflow for deploying a new application component look like?

Have a look at the Docker Hub [1].

- Do you think it was useful to create an image for Apache by ourselves?
- How are images in Docker Hub created and maintained?

[1] https://hub.docker.com/

Lesson 2: Mediawiki application with Docker

After Lesson 1, you should be familiar with Docker containers. Let's use Docker to deploy the Mediawiki application as a next step.

Task: Dockerfiles for Mediawiki

We will now use the scripts from exercise 4 (cloud-init) to deploy the mediawiki components. We will write one Dockerfile for each component (Database, Mediawiki Apache, Loadbalancer, Monitoring). We will reuse and extend existing Docker images of Docker Hub.

Create a new virtual machine with Ubuntu 16.04 and install Docker (or use the Docker VM from lesson 1). Make sure to allow incoming tcp access to port 80. Download and extract the dockerfiles.zip file from Moodle and copy it into your virtual machine. Inside the extracted folder, you find four folders, one for each component. Most sub folders contain a file called `Dockerfile`, and some other files like configuration files or the database dump. Before we continue, open the file `Mediawiki/LocalSettings.php` and change the `$wgServer` to the floating ip of your virtual machine.

Let's now build Docker images from the Docker files. Navigate into the `dockerfiles` folder of the extracted dockerfiles.zip archive.

```
docker build -t clouds/database ./Database
docker build -t clouds/mediawiki ./Mediawiki
docker build -t clouds/loadbalancer ./Loadbalancer
```

Validate that the images were created properly, via the command `docker images`.

```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
clouds/loadbalancer  latest              eb626e2194a5        2 seconds ago       109MB
clouds/mediawiki    latest              82b8228794b8        27 seconds ago      446MB
clouds/database     latest              48476b1997a0        About a minute ago  279MB
...
```

Let's start a container for each main component (database, webserver, loadbalancer). Since the mediawiki container needs to connect to the database container, and the load balancer container has to point to the mediawiki container, we have to use the `--link` argument to reference them automatically. Inside the containers, die linked container is then known by the specified name. Finally, the loadbalancer container has to export its internal port 80 as external port 80.

```
docker run -d --hostname database --name database clouds/database
docker run -d --hostname web1 --name web1 --link=database clouds/mediawiki
docker run -d --hostname web2 --name web2 --link=database clouds/mediawiki
docker run -d --hostname loadbalancer --name loadbalancer --link=web1 \
```

```
    --link=web2 --publish 80:80 clouds/loadbalancer
```

Validate with `docker ps -a` that all containers are running. Open a browser and navigate to http://YOUR_FLOATING_IP/wiki/index.php/Main_Page. You should a working Mediawiki.

Troubleshooting: if something broke, you can access the containers via `docker exec -ti web1 bash`, or get the `stdout` via `docker logs web1`.

Questions: Experiences with Docker

Since the beginning of the exercises you made experiences with virtual machines on OpenStack, automated resource allocation with Terraform and automated application deployment with cloud-init. Finally, we just "dockered" the Mediawiki example.

- Practically, where do you see benefits and drawbacks in the use of virtual machines versus Docker containers? (E.g. creation time, image size, descriptiveness, ...)
- Looking at the cloud stack from the beginning of this exercise, why are the two layers "Cloud Platform" and "Virtual Resources" still necessary although we have containers? Or why are both layers not necessary when working with containers?

Task: Extend the Monitoring

The previous task provides no monitoring. Let's extend the three existing containers by monitoring. We can use the existing Docker images for influxdb, telegraf and chronograf from Docker Hub.

```
docker run -d --hostname influxdb --name influxdb influxdb
docker run -d --hostname chronograf --name chronograf --link=influxdb \
    --publish 8888:8888 chronograf
docker run -d --hostname=telegraf --name=telegraf --link=influxdb \
    --link=web1 --link=web2 --link=loadbalancer --link=database \
    -e "HOST_PROC=/rootfs/proc" -e "HOST_SYS=/rootfs/sys" \
    -e "HOST_ETC=/rootfs/etc" \
    -v /var/run/docker.sock:/var/run/docker.sock:ro \
   -v /sys:/rootfs/sys:ro -v /proc:/rootfs/proc:ro -v /etc:/rootfs/etc:ro \
   -v $(pwd)/Monitoring/telegraf.conf:/etc/telegraf/telegraf.conf:ro \
    telegraf
```

The chronograf dashboard should be available via http://YOUR_FLOATING_IP:8888/. Make sure that your security groups allow incoming tcp connections on port 8888. The Chronograf dashboard has to be configured with connection string "http://influxdb:8086" some namy and Telegraf database "telegraf".

Task: Use docker-compose as a simple orchestrator

So far we have created and linked the Docker containers manually. Docker compose [1] can help with automation.

First, install Docker compose in your virtual machine (see [1]). Become familiar with the basic usage of Docker compose. Next, try to create a `docker-compose.yaml` file for the containers we created manually.

The empty skeleton will look like the following:

```
version: '2'
services:
    web1:
        build: Mediawiki
        image: clouds/mediawiki
        ...

    database:
        build: Database
        ...
    ...
```

The commands you would need to work with your Docker compose file are `docker-compose build`, `docker-compose up`, `docker-compose down`, ...

[1] https://docs.docker.com/compose/

Question: Docker distributed

So far we used one virtual machine to host all the Docker containers. This will of course not scale eventually.

Can you find a solution to distribute Docker containers on multiple hosts?