Learn Docker

Takács Ákos

Feb 21, 2021

INTRO:

This project contains examples and scripts to help you learn about Docker.

The examples were originally made for the participants of the Ipszilon Seminar in 2017 in Hungary. The virtual machines were created in the Cloud For Education system. Some of the scripts may not be useful to you.

Before you start working with the example projects, read *Getting started*.

CHAPTER

ONE

GETTING STARTED

1.1 Clone the git repository

```
git clone https://github.com/itsziget/learn-docker.git
```

1.2 Scripts

install.sh contains the installation of all the necessary components except the scripts below. You may need to restart your machine after the installation.

system/etc/profile.d/fixhost.sh It fixes the missing hostname in `/etc/hosts`, so you will not see error messages after using "sudo". The script checks if the machine's hostname is in the hosts file and writes into the file if the hostname was missing. In case of Ubuntu 16.04 it can copied into /etc/profile.d/.

system/usr/local/bin/xip.sh *xip.io* generates domain names for the public DNS server based on the current WAN or LAN IP address of the host machine. It must be copied into /usr/local/bin/ with the filename "xip". When you execute "xip", a domain name will be shown (Ex.: 192.168.1.2.xip.io) which you can use for the examples. The command takes one optional parameter as a subdomain. Ex.: "xip web1". The result would be: web1.192.168.1.2.xip.io

system/etc/profile.d/xip.variable.sh It uses the xip command to set the XIP environment variable so you can use the variable in a docker-compose.yml too.

Make sure you each script is executable before you continue. However, the above scripts are optional and you may not need them in a local virtual machine. If you don't want to rely on automatic IP address detection, set the XIP variable manually.

1.3 Example projects

Example projects are in the *learn-docker/projects* folder, so go to there.

Check the existence of `$XIP` variable since you will need it for some examples:

If it does not exist or empty, then set the value manually or run the script below:

All off the examples were tested with Docker 20.10.1. The version of Docker Compose was 1.27.4. You can try with more recent versions but some behaviour could be different in the future.

CHAPTER

TWO

LXD

Before using Docker containers it's good to know a little about a similar tool. LXD can run containers and also virtual machines with similar commands. It uses LXC to run containers (as Docker did at the beginning) and Qemu-KVM to run virtual machines. To install LXD 4.0 LTS you need snap.

2.1 Install LXD 4.0 LTS

```
sudo snap install --channel 4.0/stable lxd
```

Now you need to initialize the configuration:

```
lxd init
```

You will found the following questions:

1. **Questin:** Would you like to use LXD clustering? (yes/no) [default=no]:
   Answer: no
2. **Question:** Do you want to configure a new storage pool? (yes/no) [default=yes]:
   Answer: yes
3. **Question:** Name of the new storage pool [default=default]
   Answer: default
4. **Question:*** Name of the storage backend to use (btrfs, dir, lvm, zfs, ceph) [default=zfs]:
   Answer: zfs
5. **Question:** Create a new ZFS pool? (yes/no) [default=yes]:
   Answer: yes
6. **Question:** Would you like to use an existing empty block device (e.g. a disk or partition)? (yes/no) [default=no]:
   Answer: no
7. **Question:** Size in GB of the new loop device (1GB minimum) [default=25GB]:
   Answer: Choose a suitable size for you depending on how much space you have.
8. **Question:** Would you like to connect to a MAAS server? (yes/no) [default=no]:
   Answer: no
9. **Question:** Would you like to create a new local network bridge? (yes/no) [default=yes]:
   Answer: yes
10. **Question:** What should the new bridge be called? [default=lxdbr0]:

Answer: lxdbr0

11. **Question:** What IPv4 address should be used? (CIDR subnet notation, “auto” or “none”) [default=auto]:
    Answer: auto
12. **Question:** What IPv6 address should be used? (CIDR subnet notation, “auto” or “none”) [default=auto]:
    Answer: none
13. **Question:** Would you like LXD to be available over the network? (yes/no) [default=no]:
    Answer: no
14. **Question:** Would you like stale cached images to be updated automatically? (yes/no) [default=yes]
    Answer: no
15. **Question:** Would you like a YAML “lxd init” preseed to be printed? (yes/no) [default=no]:
    Answer: Optional. Type “yes” if you want to see the result of the initialization.

Output:

```
config:
  images.auto_update_interval: "0"
networks:
- config:
    ipv4.address: auto
    ipv6.address: none
  description: ""
  name: lxdbr0
  type: ""
  project: default
storage_pools:
- config:
    size: 25GB
  description: ""
  name: default
  driver: zfs
profiles:
- config: {}
  description: ""
  devices:
    eth0:
      name: eth0
      network: lxdbr0
      type: nic
    root:
      path: /
      pool: default
      type: disk
  name: default
cluster: null
```

2.2 Remote repositories

There are multiple available remote repositories to download base images. For example: https://images.linuxcontainers.org

You can list all of them with the following command:

```
lxc remote list
```

2.3 Search for images

Pass `<reponame>:<keywords>` to `lxc image list`

```
lxc image list images:ubuntu
# or
lxc image list images:ubuntu focal
# or
lxc image list images:ubuntu 20.04
# or
lxc image list ubuntu:20.04
```

2.4 Show image information

To show information about a specific image use `lxc image info` with `<reponame>:<knownalias>`

```
lxc image info ubuntu:f
```

Aliases are the names of the images with which you can refer to a specific image. One image can have multiple aliases. The previous command's output is a valid YAML so you can use yq to process it.

```
lxc image info ubuntu:focal | yq '.Aliases'
```

2.5 Start Ubuntu 20.04 container

```
lxc launch ubuntu:20.04 ubuntu-focal
```

2.6 List LXC containers

```
lxc list
```

2.7 Enter the container

```
lxc exec ubuntu-focal bash
```

Then just use `exit` to quit the container.

2.8 Delete the container

```
lxc delete --force ubuntu-focal
```

2.9 Start Ubuntu 20.04 VM

You can even create a virtual machine instead of container if you have at least LXD 4.0 installed on your machine.

```
lxc launch --vm ubuntu:20.04 ubuntu-focal-vm
```

It will not work on all machines, only when Qemu KVM is supported on that machine.

CHAPTER

THREE

DOCKER

3.1 System information

```
docker help
docker info
docker version
docker version --format '{{json .}}' | jq # requires jq installed
docker version --format '{{.Client.Version}}'
docker version --format '{{.Server.Version}}'
docker --version
```

3.2 Run a stateless DEMO application

```
docker run --rm -p "8080:80" itsziget/phar-examples:1.0
# Press Ctrl-C to quit
```

3.3 Play with the “hello-world” container

3.3.1 Start “hello-world” container

```
docker run hello-world
# or
docker container run hello-world
```

Output:

```
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
  executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.
```

(continues on next page)

(continued from previous page)

```
To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

3.3.2 List containers

List running containers

```
docker ps
# or
docker container ps
# or
docker container ls
# or
docker container list
```

List all containers

```
docker ps -a
# or use the other alias commands
```

List containers based on the hello-world image:

```
docker ps -a -f ancestor=hello-world
# or
docker container list --all --filter ancestor=hello-world
```

3.3.3 Delete containers

Delete a stopped container

```
docker rm containername
# or
docker container rm containername
```

Delete a running container:

```
docker rm -f containername
```

If the generated name of the container is “angry_shaw”

```
docker rm -f angry_shaw
```

3.3.4 Start a container with a name

```
docker run --name hello hello-world
```

Running the above command again results an error message since "hello" is already used for the previously started container. Run the following command to check the stopped containers:

```
docker ps -a
```

Or you can start the stopped container again by using its name:

```
docker start hello
```

The above command will display the name of the container. You need to start it in "attached" mode in order to see the output:

```
docker start -a hello
```

Delete the container named "hello"

```
docker rm hello
```

3.3.5 Start a container and delete it automatically when it stops

```
docker run --rm hello-world
```

3.4 Start an Ubuntu container

3.4.1 Start Ubuntu in foreground ("attached" mode)

```
docker run -it --name ubuntu-container ubuntu:20.04
```

Press `Ctrl+P` and then `Ctrl+Q` to detach from the container or type `exit` and press `enter` to exit bash and stop the container.

3.4.2 Start Ubuntu in background ("detached" mode)

Linux distribution base Docker images usually don't contain Systemd as LXD images so these containers cannot run in background unless you pass `-it` to get interactive terminal. It wouldn't be necessary with a container which has a process inside running in foreground continuously. `-it` works with other containers too as long as the containers command is "bash" or some other shell.

```
docker rm -f ubuntu-container
docker run -it -d --name ubuntu-container ubuntu:20.04
```

Note: Actually only `-i` or `-t` would be enough to keep the container in the backgorund, but if you want to attach the container later, it requires both of them. Of course, `-d` is always required.

3.4.3 Attach the container

You can attach the container and see the same as you could see when you run a container without `-d`, in foreground. You can even interact with the container's main process so be careful and don't execute a command like `exit`, or you will stop the whole container by stopping its main process.

```
docker attach ubuntu-container
```

Press Ctrl+P and then Ctrl+Q to quit without stopping the container.

The better way to "enter" the container is `docker exec` which is similar to the way of LXD.

```
docker exec -it ubuntu-container
```

Now you can use the "exit" command to quit the container and leave it running.

3.5 Start Apache HTTPD webszerver

3.5.1 Start the container in the foreground

```
docker run --name web httpd:2.4
```

There will be no prompt until you press "CTRL+C" to stop the container running in the foreground.

Note: When you change your terminal window it will send SIGWINCH signal to the container and shut down the server. Use it only for some quick test.

3.5.2 Start it in the background

```
docker rm web
docker run -d --name web httpd:2.4
```

Note: You don't need to use `-it` and you should not use that either. Running HTTPD server container with and interactive terminal will send SIGWINCH signal to the container and shut down the HTTPD server immediately when you try to attach it.

Even without `-it`, attaching the HTTPD server container will shut down the server when you change the size of your terminal window.

Use `docker logs` instead.

3.5.3 Check container logs

`docker logs` shows the standard error and output of a container without attaching it. Actually it will read and show the content of the log file which was saved from the container's output.

```
docker logs web
# or
docker container logs web
```

Watch the output (logs) continuously

```
docker logs -f web
# Press Ctrl-C to stop watching
```

3.5.4 Open the webpage using an IP address

Get the IP address:

```
CONTAINER_IP=$(docker container inspect web --format '{{.NetworkSettings.IPAddress}}')
```

You can test if the server is working using wget:

```
wget -qO- $CONTAINER_IP
```

Output:

```
<html><body><h1>It works!</h1></body></html>
```

3.5.5 Use port forwarding

Delete the container named "web" and forward the port 8080 from the host to the containers internal port 80:

```
docker rm -f web
docker run -d -p "8080:80" --name web httpd:2.4
```

Then you can access the page using the host's IP address.

3.5.6 How we could enter a container in the past

Before `docker exec` was introduced, `nsenter` was the only way to enter a container. It does almost the same as `docker exec` except it does not support Pseudo-TTY so some commands may not work.

```
CONTAINER_PID=$(docker container inspect --format '{{ .State.Pid }}' web)

sudo nsenter \
  --target $CONTAINER_PID \
  --mount \
  --uts \
  --ipc \
  --net \
  --pid \
  --cgroup \
```

(continues on next page)

(continued from previous page)

```
  --wd \
  env -i - $(sudo cat /proc/$CONTAINER_PID/environ | xargs -0) bash
```

As you can see, `nsenter` runs a process inside specific Linux namespaces.

3.5.7 Share namespaces

```
docker rm -f web
docker run -d --name web \
  --net host \
  --uts host \
  --pid host \
  httpd:2.4
```

This example shows how you can share the host's namespaces with the container.

- **net**: The container will not get a virtual network. Localhost inside the container will the same as localhost on the host operating system.
- **uts**: When you enter the container you will see that the hostname in the prompt is the same as you can see on the host. Without this, the container had a random hash as hostname.
- **pid**: The container can see every process running on the host and not just inside the container.

Note: Using user namespace in a Docker container is disabled by default

---

Now enter the container

```
docker exec -it web bash
```

and install the following tools, so you can see host processes and network interfaces from the container.

```
apt update
apt install iproute2 procps psmisc
```

- **iproute2**: adds the `ip` command
- **procps**: installs the `ps` command
- **psmisc**: this makes `pstree` command available

Now run

- `ip addr` to see network interfaces
- `ps auxf` to see host processes
- `pstree` to see the process tree

You can exit the container and run the following command to get only the processes inside the container:

```
docker exec web ps auxf $(docker container inspect --format '{{ .State.Pid }}' web)
```

3.6 Start Ubuntu virtual machine

There are multiple ways to run a virtual machine with Docker. Using a parameter is not enough. You need to choose a different runtime. The default is `runc` which runs containers. One of the most popular and easiest runtime is Kata Containers.

Follow the instructions to install the latest stable version of the Kata runtime

Source: Install Kata Containers on Ubuntu

```
ARCH=$(arch)
BRANCH="1.12"
sudo sh -c "echo 'deb http://download.opensuse.org/repositories/home:/katacontainers:/
↪releases:/${ARCH}:/${BRANCH}/xUbuntu_$(lsb_release -rs)/ /' > /etc/apt/sources.list.
↪d/kata-containers.list"
curl -sL  http://download.opensuse.org/repositories/home:/katacontainers:/releases:/$
↪{ARCH}:/${BRANCH}/xUbuntu_$(lsb_release -rs)/Release.key | sudo apt-key add -
sudo -E apt-get update
sudo -E apt-get -y install kata-runtime kata-proxy kata-shim
```

and configure Docker daemon to use it. An example `/etc/docker/daemon.json` is the following:

```
{
  "default-runtime": "runc",
  "runtimes": {
    "kata": {
      "path": "/usr/bin/kata-runtime"
    }
  }
}
```

Now run

```
docker run -d -it --runtime kata --name ubuntu-vm ubuntu:20.04
```

It is still lightweight. You can run `ps aux` inside to see there is no systemd or other process like that, however, run the following command on the host machine and see it has only one CPU core:

```
docker exec -it ubuntu-vm cat /proc/cpuinfo
```

CHAPTER

FOUR

START A SIMPLE WEB SERVER WITH MOUNTED DOCUMENT ROOT

Go to Project 1 from the git repository root:

```
cd projects/p01
```

Start the container:

```
docker run -d -v $(pwd)/www:/usr/local/apache2/htdocs:ro --name p01_httpd -p "8080:80
↪" httpd:2.4
# or
docker run -d --mount type=bind,source=$(pwd)/www,target=/usr/local/apache2/htdocs,
↪readonly --name p01_httpd -p "8080:80" httpd:2.4
```

Generate a domain name:

```
xip
# example output:
# 192.168.1.2.xip.io
```

Test the web page:

```
http://192.168.1.2.xip.io:8080
# output:
# Hello Docker (p01)
```

Delete the container to make port 8080 free again.

```
docker rm -f p01_httpd
```

CHAPTER

FIVE

BUILD YUR OWN WEB SERVER IMAGE AND COPY THE DOCUMENT ROOT INTO THE IMAGE

Go to Project 2 from the git repository root:

```
cd projects/p02
```

Building an image:

```
docker build -t localhost/p02_httpd .
```

The dot character at the and of the line is important and required.

Start container:

```
docker run -d --name p02_httpd -p "80:80" localhost/p02_httpd
```

You can open the website from a web browser on port 80. The output should be “Hello Docker (p02)”

Delete the container to make port 8080 free again.

```
docker rm -f p02_httpd
```

CHAPTER

SIX

CREATE YOUR OWN PHP APPLICATION WITH BUILT-IN PHP WEB SERVER

Go to Project 3 from the git repository root:

```
cd projects/p03
```

Build an image:

```
docker build -t localhost/p03_php .
```

Start the container:

```
docker run -d --name p03_php -p "8080:80" localhost/p03_php
```

Open in a web browser and reload the page multiple times. You can see the output is different each time with more lines.

Now delete the container. Probably you already now how, but as a reminder I show you again:

```
docker rm -f p03_php
```

Execute the "docker run . . . " command again and reload the example web page to see how you have lost the previously generated lines and delete the container again.

Now start the container with a volume to preserve data:

```
docker run -d --mount source=p03_php_www,target=/var/www --name p03_php -p "8080:80"␣
↪localhost/p03_php
```

This way you can delete and create the container repeatedly a you will never lose your data until you delete the volume. You can see all volumes with the following command:

```
docker volume ls
# or
docker volume list
```

After you have deleted the container, you can delete the volume:

```
docker volume rm p03_php_www
```

CHAPTER

SEVEN

CREATE A SIMPLE DOCKER COMPOSE PROJECT

Go to Project 4 from the git repository root:

```
cd projects/p04
```

Build an image and start the container using Docker Compose:

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

Check the container:

```
docker-compose ps
# The name of the container: p04_php_1
```

Check the networks:

```
docker network ls
# New bridge network: p04_default
```

Delete the container, and networks with Docker Compose:

```
docker-compose down
```

Or delete the volumes too.

```
docker-compose down --volume
```

docker-compose down

CHAPTER

EIGHT

COMMUNICATION OF PHP AND APACHE HTTPD WEB SERVER WITH THE HELP OF DOCKER COMPOSE

Go to Project 5 from the git repository root:

```
cd projects/p05
```

Build PHP image and start the containers:

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

Start multiple container for PHP:

```
docker-compose up -d --scale php=2
```

List the containers to see PHP has multiple instance:

```
docker-compose ps
```

Open the page in your browser and you can see the hostname in the first line is not constant. It changes but not every time, although the data is permanent.

Delete everything created by Docker Compose for this project:

```
docker-compose down --volume
```

CHAPTER

NINE

RUN MORE DOCKER COMPOSE PROJECT ON THE SAME PORT USING NGINX-PROXY

See nginx-proxy

Go to Project 6 from the git repository root:

```
cd projects/p06
```

Create the proxy network:

```
docker network create public_proxy
```

Check the networks:

```
docker network ls
```

Navigate to the nginxproxy folder

```
cd nginxproxy
```

Start the proxy:

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

Navigate to the web1 folder:

```
cd ../web1
```

At this point you need to have the XIP variable set as *Welcome to Learn Docker's documentation!* refers to it.

Alternative option: set the XIP variable in the ".env" file:

Start the containers:

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

Navigate to the web2 folder:

```
cd ../web2
```

Start the containers:

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

Both of the services are available on port 80. Example:

```
http://web1.192.168.1.6.xip.io
http://web2.192.168.1.6.xip.io
```

This way you do not need to remove a container just because it is running on the same port you want to use for a new container.

Clean the project:

```
docker-compose down --volume
cd ../web1
docker-compose down --volume
cd ../nginxproxy
docker-compose down --volume
```

CHAPTER

TEN

PROTECT YOUR WEB SERVER WITH HTTP AUTHENTICATION

Go to Project 7 from the git repository root:

```
cd projects/p07
```

The first step is the same as it was in *Run more Docker Compose project on the same port using nginx-proxy*. Start the proxy server:

```
cd nginxproxy
docker-compose up -d
```

Go to the web folder:

```
cd ../web
```

You can simply start a web server protected by HTTP authentication. The name and the password will come from environment variables. I recommend you to use a more secure way in production. Create the .htpasswd file manually and mount it inside the container.

The htpasswd container will create .htpasswd automatically and exit.

In the ".env" file you can find two variables. HTTPD_USER and HTTPD_PASS will be used in "docker-compose.yml" by the "htpasswd" service to generate the password file and then the "httpd" service will read it from the common volume.

The "fixperm" service runs and exits similar to "htpasswd". It sets the permission of the files after the web server starts.

Use the "depends_on" option to control which service starts first.

At this point you need to have the XIP variable set as the *Welcome to Learn Docker's documentation!* refers to it.

Alternative option: set the XIP variable in the ".env" file:

Start the web server

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

Open the web page in your browser (Ex.: p07.192.168.1.6.xip.io). You will get a password prompt.

Clean the project:

```
docker-compose down --volume
cd ../nginxproxy
docker-compose down --volume
```

CHAPTER

ELEVEN

MEMORY LIMIT TEST WITH PHP IN A CONTAINER

Go to Project 8 from the git repository root:

```
cd projects/p08
```

11.1 Description

This example shows the memory testing in a PHP container, where the "truncate" command generates a file with a defined size and the PHP reads it into the memory. We use an environment variable to set the memory size.

11.2 Start the test

The container will have 50MB memory limit. (It must be at least 6MB in Docker Compose 1.27.4). The examples below will test the memory usage from 10MB to 50MB increased by 10MB for each test.

```
MEMSIZE=10MiB docker-compose run --rm php
MEMSIZE=20MiB docker-compose run --rm php
MEMSIZE=30MiB docker-compose run --rm php
MEMSIZE=40MiB docker-compose run --rm php
MEMSIZE=50MiB docker-compose run --rm php
```

output:

```
bash: line 5:     9 Killed                  php -r '
   ob_start();
   readfile("/tmp/50MiB");
   ob_clean();
   echo (memory_get_peak_usage(true)/1024/1024)." MiB\n";
'
```

"Killed" means we exceeded the memory limit. There is no error until 50MB. Since there is some additional memory usage in the container, it kills the process at 50MiB even though 50 is still allowed.

11.3 Explanation of the parameters

The “docker-compose run” is similar to “docker run”, but it runs a service from the compose file. “–rm” means the same as it meant for “docker run”. Deletes the container right after it stopped.

Clean the project:

```
docker-compose down
```

The containers were deleted automatically, but it can still delete the network.

CHAPTER

TWELVE

CPU LIMIT TEST

We test the CPU limit in this example using an image based on petermaric/docker.cpu-stress-test. Since that image is outdated, we create a new image similar to Peter Maric's work.

```
docker build -t localhost/stress .
```

Execute the following command to test a whole CPU core:

```
docker run -it --rm \
  -e STRESS_MAX_CPU_CORES=1 \
  -e STRESS_SYSTEM_FOR=30 \
  --cpus=1 \
  localhost/stress
```

Run "top" in an other terminal to see that the "stress" process uses 100% of one CPU.

Press Ctrl-C and execute the following command to test two CPU core and allow the container to use only 1 and a half CPU.

```
docker run -it --rm \
  -e STRESS_MAX_CPU_CORES=2 \
  -e STRESS_TIMEOUT=30 \
  --cpus=1.5 \
  localhost/stress
```

Use "top" again to see that the "stress" process uses 75% of two CPU.

You can test on one CPU core again and allow the container to use 50% of a specific CPU core by setting the core index.

```
docker run -it --rm \
  -e MAX_CPU_CORES=1 \
  -e STRESS_SYSTEM_FOR=60 \
  --cpus=0.5 \
  --cpuset-cpus=0 \
  localhost/stress
```

You can use top again, but do not forget to add the index column to the list:

- run "top"
- press "f"
- Select column "P" by navigating with the arrow keys
- Press "SPACE" to select "P"
- Press "ESC"

Now you can see the indexes in the column “P”.

Press “1” to list all the CPU-s at the top of the terminal so you can see the usage of all the CPU-s.

Clean the project:

```
docker-compose down
```