TD CONTAINERS LXD-DOCKER

1 Preparation

1. Create and launch a full VM with QEMU.

2 Installing LXD, the Linux container daemon

1. First, update the apt cache and try to install LXD. This should install updates to the LXD package, if any:

```
$ sudo apt-get update
$ sudo apt-get install lxd
```

2. Along with LXD, we will need one more package named ZFS—the most important addition to Ubuntu 16.04. We will be using ZFS as a storage backend for LXD:

```
$ sudo apt-get install zfsutils-linux
```

3. Once LXD has been installed, we need to configure the daemon before we start using it. Use lxd init to start the initialization process. This will ask some questions about the LXD configuration:

```
$ sudo lxd init
Name of the storage backend to use (dir or zfs): zfs
Create a new ZFS pool (yes/no)? yes
Name of the new ZFS pool: lxdpool
Would you like to use an existing block device (yes/no)? no
Size in GB of the new loop device (1GB minimum): 10
Would you like LXD to be available over the network (yes/no)?
no
Do you want to configure the LXD bridge (yes/no)? yes
Warning: Stopping lxd.service, but it can still be activated
by: lxd.socket
LXD has been successfully configured.
```

Now, we have our LXD setup configured and ready to use. In the next recipe, we will start our first container with LXD.

3 Deploying your first container with LXD

LXD works on the concept of remote servers and images served by those remote servers. Starting a new container with LXD is as simple as downloading a container image and starting a container out of it, all with a single command.
Follow these steps:

1. To start your first container, use the lxc launch command, as follows:

```
$ lxc launch ubuntu:14.04/amd64 c1
```

LXC will download the required image (14.04/amd64) and start the container.

2. As you can see in the screenshot, lxc launch downloads the required image, creates a new container, and then starts it as well. You can see your new container in a list of containers with the lxc list command, as follows:

```
$ lxc list
```

3. Optionally, you can get more details about the containers with the lxc info command:

```
$ lxc info c1
```

4. Now that your container is running, you can start working with it. With the lxc exec command, you can execute commands inside a container. Use the following command to obtain the details of Ubuntu running inside a container:

```
$ lxc exec c1 -- lsb_release -a
```

5. You can also open a bash shell inside a container, as follows:

```
$ lxc exec c1 – bash
```

Creating images is a time-consuming task. With LXD, the team has solved this problem by downloading the prebuilt images from trusted remote servers. Unlike LXC, where images are built locally, LXD downloads them from the remote servers and keep a local cache of these images for later use.
The default installation contains three remote servers:

- Ubuntu: This contains all Ubuntu releases
- Ubuntu-daily: This contains all Ubuntu daily builds
- images: This contains all other Linux distributions

You can get a list of available remote servers with this command:

```
$ lxc remote list
```

Similarly, to get a list of available images on a specific remote server, use the following command:

```
$ lxc image list ubuntu:
```

The lxc launch command creates a new container and then starts it as well. If you want to just create a container without starting it, you can do that with the lxc init command, as follows:

```
$ lxc init ubuntu:xenial c2
```

All containers (or their rootfs) are stored under the `/var/lib/lxd/containers` directory, and images are stored under the `/var/lib/lxd/images` directory.

While starting a container, you can specify the set of configuration parameters using the

--config flag. LXD also supports configuration profiles.
Profiles are a set of configuration parameters that can be applied to a group of containers. Additionally, a container can have multiple profiles. LXD ships with two preconfigured profiles: `default` and `docker` .

To get a list of profiles, use the `lxc profile list` command, and to get the contents of a profile, use the `lxc profile show <profile_name>` command
.
Sometimes, you may need to start a container to experiment with something—execute a few random commands and then undo all the changes. LXD allows us to create such throwaway or ephemeral containers with the -e flag. By default, all LXD containers are permanent containers. You can start an ephemeral container using the --ephemeral or -e flag. When stopped, an ephemeral container will be deleted automatically.

With LXD, you can start and manage containers on remote servers as well. For this, the LXD daemon needs to be exposed to the network. This can be done at the time of initializing LXD or with the following commands:

```
$ lxc config set core.https_address "[::]"
$ lxc config set core.trust_password some-password
```

Next, make sure that you can access the remote server and add it as a remote for LXD with the lxc remote add command:

```
$ lxc remote add remote01 192.168.0.11 # lxc remote add name server_ip
```

Now, you can launch containers on the remote server, as follows:

```
$ lxc launch ubuntu:xenial remote01:c1
```

Unlike LXC, LXD container images do not support password-based SSH logins.
The container still has the SSH daemon running, but login is restricted to a public key. You need to add a key to the container before you can log in with SSH.
LXD supports file management with the lxc file command; use it as follows to set your public key inside an Ubuntu container:

```
$ lxc file push ~/.ssh/id_rsa.pub \
c1/home/ubuntu/.ssh/authorized_keys \
--mode=0600 --uid=1000
```

Once the public key is set, you can use SSH to connect to the container, as follows:

```
$ ssh ubuntu@container_IP
```

Alternatively, you can directly open a root session inside a container and get a bash shell with lxc exec , as follows:

```
$ lxc exec c1 – bash
```

4 Managing LXD containers

Follow these steps to manage LXD containers:

1. Before we start with container management, we will need a running container. If you have been following the previous recipes, you should already have a brand new container running on your system. If your container is not already running, you can start it with the lxc start command:

```
$ lxc start c1
```

2. To check the current state of a container, use lxc list , as follows:

```
$ lxc list c1
```

This command should list only containers that have `c1` in their name.

3. You can also set the container to start automatically. Set the boot.autostart configuration option to true and your container will start automatically on system boot. Additionally, you can specify a delay before autostart and a priority in the autostart list:

```
$ lxc config set c1 boot.autostart true
```

4. Once your container is running, you can open a bash session inside a container using the lxc exec command:

```
$ lxc exec c1 -- bash
root@c1:~# hostname
c1
```

This should give you a root shell inside a container. Note that to use bash, your container image should have a bash shell installed in it. With alpine containers, you need to use sh as the shell as alpine does not contain the bash shell.

5. LXD provides the option to pause a container when it's not being actively used. A paused container will still hold memory and other resources assigned to it, but not receive any CPU cycles:

```
$ lxc pause c1
```

6. Containers that are paused can be started again with `lxc start`.

7. You can also restart a container with the lxc restart command, with the option to perform a stateful or stateless restart:

```
$ lxc restart --stateless c1
```

8. Once you are done working with the container, you can stop it with the lxc stop command. This will release all resources attached to that container:

```
$ lxc stop c1
```

At this point, if your container is an ephemeral container, it will be deleted automatically.

9. If the container is no longer required, you can explicitly delete it with the lxc delete command:

```
$ lxc delete c1
```

Follow these steps to deal with LXD containers:

1. Sometimes, you may need to clone a container and have it running as a separate system. LXD provides a copy command to create such clones:

```
$ lxc copy c1 c2 # lxc copy source destination
```

You can also create a temporary copy with the `--ephemeral` flag and it will be deleted after one use.

2. Similarly, you can create a container, configure it as per you requirements, have it stored as an image, and use it to create more containers. The `lxc publish` command allows you to export existing containers as a new image. The resulting image will contain all modifications from the original container:

```
$ lxc publish c1 --alias nginx # after installing nginx
```

The container to be published should be in the stopped state. Alternatively, you can use the `--force` flag to publish a running container, which will internally stop the container before exporting.

3. You can also move the entire container from one system to another. The move command helps you with moving containers across hosts. If you move a container on the same host, the original container will be renamed. Note that the container to be renamed must not be running:

```
$ lxc move c1 c2 # container c1 will be renamed to c2
```

4. Finally, we have the snapshot and restore functionality. You can create snapshots of the container or, in simple terms, take a backup of its current state. The snapshot can be a stateful snapshot that stores the container's memory state. Use the following command to create a snapshot of your container:

```
$ lxc snapshot c1 snap1 # lxc snapshot container cnapshot
```

5. The lxc list command will show you the number of snapshots for a given container. To get the details of every snapshot, check the container information with the lxc info command:

```
$ lxc info c1
…
Snapshots:
    c1/shap1 (taken at 2016/05/22 10:34 UTC) (stateless)
```

6. Once you have the snapshots created, you can restore it to go back to a point or create new containers out of your snapshots and have both states maintained. To restore your snapshot, use lxc restore , as follows:

```
$ lxc restore c1 snap1 # lxc restore container snapshot
```

7. To create a new container out of your snapshot, use lxc copy , as follows:

```
$ lxc copy c1/snap1 c4 # lxc copy container/snapshot
new_container
```

8. When you no longer need a snapshot, delete it with lxc delete , as follows:

```
$ lxc delete c1/snap1 # lxc delete container/snapshot
```

5 Setting resource limits on LXD containers

In this recipe, we will learn to set resource limits on containers. LXD uses the cgroups feature in the Linux kernel to manage resource allocation and limits. Limits can be applied to a single container through configuration or set in a profile, applying limits to a group of containers at once. Limits can be dynamically updated even when the container is running.

We will create a new profile and configure various resource limits in it. Once the profile is ready, we can use it with any number of containers.
Follow these steps:

1. Create a new profile with the following command:

```
$ lxc profile create cookbook
Profile cookbook created
```

2. Next, edit the profile with lxc profile edit . This will open a text editor with a default profile structure in YML format:

```
$ lxc profile edit cookbook
```

Add the following details to the profile. Feel free to select any parameters and change their values as required:

```
name: cookbook
config:
    boot.autostart: "true"
    limits.cpu: "1"
    limits.cpu.priority: "10"
    limits.disk.priority: "10"
    limits.memory: 128MB
    limits.processes: "100"
description: A profile for Ubuntu Cookbook Containers
devices:
    eth0:
        nictype: bridged
        parent: lxdbr0
        type: nic
```

Save your changes to the profile and exit the text editor.

3. Optionally, you can check the created profile, as follows:

```
$ lxc profile show cookbook
```

4. Now, our profile is ready and can be used with a container to set limits. Create a new container using our profile:

```
$ lxc launch ubuntu:xenial c4 -p cookbook
```

5. This should create and start a new container with the cookbook profile applied to it. You can check the profile in use with the lxc info command:

```
$ lxc info c4
```

6. Check the memory limits applied to container c4 :

```
$ lxc exec c4 -- free -m
```

7. Profiles can be updated even when they are in use. All containers using that profile will be updated with the respective changes, or return a failure message. Update your profile as follows:

```
$ lxc profile set cookbook limits.memory 256MB
```

In the previous example, we used the edit option to edit the profile and set multiple parameters at once. You can also set each parameter separately or update the profile with the set option:

```
$ lxc profile set cookbook limits.memory 256MB
```

Similarly, use the get option to read any single parameter from a profile:

```
$ lxc profile get cookbook limits.memory
```

Profiles can also be applied to a running container with lxc profile apply . The following command will apply two profiles, default and cookbook , to an existing container, c6 :

```
$ lxc profile apply c6 default,cookbook
```

Updating the profiles will update the configuration for all container using that profile. To modify a single container, you can use lxc config set or pass the parameters directly to a new container using the -c flag:

```
$ lxc launch ubuntu:xenial c7 -c limits.memory=64MB
```

Similar to lxc profile , you can use the edit option with lxc config to modify multiple parameters at once. The same command can also be used to configure or read server parameters. When used without any container name, the command applies to the LXD daemon.

The lxc profile and lxc config commands can also be used to attach local devices to containers. Both commands provide the option to work with various devices, which include network, disk IO, and so on. The simplest example will be to pass a local directory to a container, as follows:

```
$ lxc config device add c1 share disk \
source=/home/ubuntu path=home/ubuntu/shared
```

6 Networking with LXD

By default, LXD sets up a NAT network for containers. This is a private network attached to the

lxdbr0 port on the host system. With this setup, containers get access to the Internet, but the containers themselves or the services running in the containers are not accessible from an outside network. To open a container to an external network, you can either set up port forwarding or use a bridge to attach the container directly to the host's network:

1. To set up port forwarding, use the iptables command, as follows:

```
$ sudo iptables -t nat -A PREROUTING -p tcp -i eth0 \
--dport 80 -j DNAT --to 10.106.147.244:80
```

This will forward any traffic on the host TCP port 80 to the containers' TCP port 80 with the IP 10.106.147.244 . Make sure that you change the port and IP address as required.

2. You can also set a bridge that connects all containers directly to your local network. The bridge will use an Ethernet port to connect to the local network. To set a bridge network with the host, we first need to create a bridge on the host and then configure the container to use that bridge adapter.
To set up a bridge on the host, open the /etc/network/interfaces file and add the following lines:

```
auto br0
iface br0 inet dhcp
    bridge_ports eth0
```

Make sure that you replace eth0 with the name of the interface connected to the external network.

3. Enable IP forwarding under sysctl .
Find the following line in /etc/sysctl.conf and uncomment it:

```
net.ipv4.ip_forward=1
```

4. Start a new bridge interface with the ifup command:

```
$ sudo ifup br0
```

5. If required, you can restart the networking service, as follows:

```
$ sudo service networking restart
```

6. Next, we need to update the LXD configuration to use our new bridge interface.

Execute a reconfiguration of the LXD daemon and choose <No> when asked to create a new bridge :

```
$ sudo dpkg-reconfigure -p medium lxd
```

7. Then on the next page, choose <Yes> to use an existing bridge.

8. Enter the name of the newly created bridge interface: <br0>

This should configure LXD to use our own bridge network and skip the internal bridge.
You can check the new configuration under the default profile:

```
$ lxc profile show default
```

9. Now, start a new container. It should receive the IP address from the router on your local network. Make sure that your local network has DHCP configured:

```
$ lxc list
```

By default, LXD sets up a private network for all containers. A separate bridge, lxdbr0 , is set up and configured in the default profile. This network is shared (NAT) with the host system, and containers can access the Internet through this network. In the previous example, we used IPtables port forwarding to make the container port 80 available on the external network. This way, containers will still use the same private network, and a single application will be exposed to the external network through the host system. All incoming traffic on host port 80 will be directed to the container's port 80 .

You can also set up your own bridge connected to the physical network. With this bridge, all your containers can connect to and be directly accessible over your local network. Your local DHCP will be used to assign IP addresses to containers. Once you create a bridge, you need to configure it with LXD containers either through profiles or separately with container configuration. In the previous example, we reconfigured the LXD network to set a new bridge.

If you are using virtual machines for hosting containers and want to set up a bridge, then make sure that you have enabled promiscuous mode on the network adapter of the virtual machine. This can be enabled from the network settings of your hypervisor. Also, a bridge setup may not work if your physical machine is using a wireless network.

LXD supports more advanced network configuration by attaching the host eth interface directly to a container. The following settings in the container configuration will set the network type to a physical network and use the host's eth0 directly inside a container. The eth0 interface will be unavailable for the host system till the container is live:

```
$ lxc config device add c1 eth0 nic nictype=physical parent=eth0
```

LXD creates a default bridge with the name lxdbr0 . The configuration file for this bridge is located at /etc/default/lxd-bridge . This file contains various configuration parameters, such as the address range for the bridge, default domain, and bridge name. An interesting parameter is the additional configuration path for dnsmasq configurations.
The LXD bridge internally uses dnsmasq for DHCP allocation. The additional configuration file can be used to set up various dnsmasq settings, such as address reservation and name resolution for containers.

Edit /etc/default/lxd-bridge to point to the dnsmasq configuration file:

```
# Path to an extra dnsmasq configuration file
LXD_CONFILE="/etc/default/dnsmasq.conf"
```

Then, create a new configuration file called /etc/default/dnsmasq.conf with the following contents:

```
dhcp-host=c5,10.71.225.100
server=/lxd/10.71.225.1
#interface=lxdbr0
```

This will reserve the IP 10.71.225.100 for the container called c5 , and you can also ping containers with that name, as follows:

```
$ ping lxd.c5
```

7 Installing Docker

Docker is an application container designed to package and run a single service. It enables developers to enclose an app with all dependencies in an isolated container environment. Docker helps developers create a reproducible environment with a simple configuration file called a Dockerfile. It also provides portability by sharing the Dockerfile, and developers can be sure that their setup will work the same on any system with the Docker runtime.

Docker is very similar to LXC. Its development started as a wrapper around the LXC API to help DevOps take advantage of containerization. It added some restrictions to allow only a single process to be running in a container, unlike a whole operating system in LXC. In subsequent versions, Docker changed its focus from LXC and started working on a new standard library for application containers, known as libcontainer.

It still uses the same base technologies, such as Linux namespaces and control groups, and shares the same kernel with the host operating system. Similarly, Docker makes use of operating system images to run containers. Docker images are a collection of multiple layers, with each layer adding something new to the base layer. This something new can include a service, such as a web server, application code, or even a new set of configurations. Each layer is independent of the layers above it and can be reused to create a new image. Being an application container, Docker encourages the use of a microservice-based distributed architecture. Think of deploying a simple WordPress blog. With Docker, you will need to create at least two different containers, one for the MySQL server and the other for the WordPress code with PHP and the web server. You can separate PHP and web servers in their own containers. While this looks like extra effort, it makes your application much more flexible. It enables you to scale each component separately and improves application availability by separating failure points.

While both LXC and Docker use containerization technologies, their use cases are different. LXC enables you to run an entire lightweight virtual machine in a container, eliminating the inefficiencies of virtualization. Docker enables you to quickly create and share a self-dependent package with your application, which can be deployed on any system running Docker.

Recently, Docker released version 1.11 of the Docker engine. We will follow the installation steps provided on the Docker site to install the latest available version:

1. First, add a new gpg key:

```
$ sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 –recv-keys \
58118E89F3A912897C070ADBF76221572C52609D
```

2. Next, add a new repository to the local installation sources. This repository is maintained by Docker and contains Docker packages for 1.7.1 and higher versions:

```
$ echo "deb https://apt.dockerproject.org/repo ubuntu-xenial main" | \
sudo tee /etc/apt/sources.list.d/docker.list
```

3. Next, update the apt package list and install Docker with the following commands:

```
$ sudo apt-get update
$ sudo apt-get install docker-engine
```

4. Once the installation completes, you can check the status of the Docker service, as follows:

```
$ sudo service docket status
```

5. Check the installed Docker version with docker version :

```
$ sudo docker version
Client:
Version: 1.11.1
API version: 1.23
...
Server:
Version: 1.11.1
API version: 1.23
…
```

6. Download a test container to test the installation. This container will simply print a welcome message and then exit:

```
$ sudo docker run hello-world
```

7. At this point, you need to use sudo with every Docker command. To enable a non-sudo user to use Docker, or to simply avoid the repeated use of sudo , add the respective usernames to the docker group:

```
$ sudo gpasswd -a ubuntu docker
```

Now, update group membership, and you can use Docker without the sudo command:

```
$ newgrp docker
```

Optional : Docker provides a quick installation script, which can be used to install Docker with a single command. This scripts reads the basic details of your operating system, such as the distribution and version, and then executes all the required steps to install Docker. You can use the bootstrap script as follows:

```
$ sudo curl -sSL https://get.docker.com | sudo sh
```

8 Starting and managing Docker containers

Let's create a new Docker container and start it. With Docker, you can quickly start a container with the docker run command:

1. Start a new Docker container with the following command:

```
$ docker run -it --name dc1 ubuntu /bin/bash
Unable to find image 'ubuntu:trusty' locally
trusty: Pulling from library/ubuntu
6599cadaf950: Pull complete
23eda618d451: Pull complete
```

```
...
Status: Downloaded newer image for ubuntu:trusty
root@bd8c99397e52:/#
```

Once a container has been started, it will drop you in a new shell running inside it. From here, you can execute limited Ubuntu or general Linux commands, which will be executed inside the container.

2. When you are done with the container, you can exit from the shell by typing exit or pressing Ctrl + D. This will terminate your shell and stop the container as well.

3. Use the docker ps command to list all the containers and check the status of your last container:

```
$ docker ps -a
```

By default, docker ps lists all running containers. As our container is no longer running, we need to use the -a flag to list all available containers.

4. To start the container again, you can use the docker start command. You can use the container name or ID to specify the container to be started:

```
$ docker start -ia dc1
```

The -i flag will start the container in interactive mode and the -a flag will attach to a terminal inside the container. To start a container in detached mode, use the start command without any flags. This will start the container in the background and return to the host shell:

```
$ docker start dc1
```

5. You can open a terminal inside a detached container with docker attach :

```
$ docker attach dc1
```

6. Now, to detach a terminal and keep the container running, you need the key combinations Ctrl + P and Ctrl + Q. Alternatively, you can type exit or press Ctrl + C to exit the terminal and stop the container.

7. To get all the details of a container, use the docker inspect command with the name or ID of the container:

```
$ docker inspect dc1 | less
```

This command will list all the details of the container, including container status, network status and address, and container configuration files.

8. To execute a command inside a container, use docker exec . For example, the following command gets the environment variables from the dc1 container:

```
$ docker exec dc1 env
```

This one gets the IP address of a container:

```
$ docker exec dc1 ifconfig
```

9. To get the processes running inside a container, use the docker top command:

```
$ docker top dc1
```

10. Finally, to stop the container, use docker stop , which will gracefully stop the container after stopping processes running inside it:

```
$ docker stop dc1
```

11. When you no longer need the container, you can use docker rm to remove/delete it:

```
$ docker rm dc1
```

9 Creating images with a Dockerfile

Dockerfiles help in automating identical and repeatable image creation. They contain multiple commands in the form of instructions to build a new image. These instructions are then passed to the Docker daemon through the docker build command. The Docker daemon independently executes these commands one by one. The resulting images are committed as and when necessary, and it is possible that multiple intermediate images are created. The build process will reuse existing images from the image cache to speed up build process.

1. First, create a new empty directory and enter it. This directory will hold our Dockerfile:

```
$ mkdir myimage
$ cd myimage
```

2. Create a new file called Dockerfile :

```
$ touch Dockerfile
```

3. Now, add the following lines to the newly created file. These lines are the instructions to create an image with the Apache web server. We will look at more details later in this recipe:

```
FROM ubuntu:trusty
MAINTAINER ubuntu server cookbook
# Install base packages
RUN apt-get update && apt-get -yq install apache2 && \
apt-get clean && \
rm -rf /var/lib/apt/lists/*
RUN echo "ServerName localhost" >>
/etc/apache2/apache2.conf
ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2
ENV APACHE_PID_FILE /var/run/apache2.pid
ENV APACHE_LOCK_DIR /var/www/html
VOLUME ["/var/www/html"]
EXPOSE 80
CMD ["/usr/sbin/apache2", "-D", "FOREGROUND"]
```

4. Save the changes and start the docker build process with the following command:

```
$ docker build
```

This will build a new image with Apache server installed on it. The build process will take a little longer to complete and output the final image ID:

5. Once the image is ready, you can start a new container with it:

```
$ docker run -p 80:80 -d image_id
```

Replace image_id with the image ID from the result of the build process.

6. Now, you can list the running containers with the docker ps command. Notice the ports column of the output:

```
$ docker ps
```

Apache server's default page should be accessible at your host domain name or IP address.

A Dockerfile is a document that contains several commands to create a new image. Each command in a Dockerfile creates a new container, executes that command on the new container, and then commits the changes to create a new image. This image is then used as a base for executing the next command. Once the final command is executed, Docker returns the ID of the final image as an output of the docker build command.
This recipe demonstrates the use of a Dockerfile to create images with the Apache web server. The Dockerfile uses a few available instructions. As a convention, the instructions file is generally called Dockerfile. Alternatively, you can use the -f flag to pass the instruction file to the Docker daemon. A Dockerfile uses the following format for instructions:
comment
INSTRUCTION argument
All instructions are executed one by one in a given order. A Dockerfile must start with the FROM instruction, which specifies the base image to be used. We have started our Dockerfile with Ubuntu:trusty as the base image. The next line specifies the maintainer or the author of the Dockerfile, with the MAINTAINER instruction.

Followed by the author definition, we have used the RUN instruction to install Apache on our base image. The RUN instruction will execute a given command on the top read-write layer and then commit the results. The committed image will be used as a starting point for the next instruction. If you've noticed the RUN instruction and the arguments passed to it, you can see that we have passed multiple commands in a chained format. This will execute all commands on a single image and avoid any cache-related problems. The apt-get clean and rm commands are used to remove any unused files and minimize the resulting image size.
After the RUN command, we have set some environment variables with the ENV instruction. When we start a new container from this image, all environment variables are exported to the container environment and will be accessible to processes running inside the container. In this case, the process that will use such a variable is the Apache server.
Next, we have used the VOLUME instruction with the path set to /var/www/html . This instruction creates a directory on the host system, generally under Docker root, and mounts it inside the container on the specified path. Docker uses volumes to decouple containers from the data they create. So even if the container using this volume is removed, the data will persist on the host system. You can specify volumes in a Dockerfile or in the command line while running the container, as follows:

```
$ docker run -v /var/www/html image_id
```

You can use docker inspect to get the host path of the volumes attached to container.

Finally, we have used the EXPOSE instruction, which will expose the specified container port to the host. In this case, it's port 80 , where the Apache server will be listening for web requests. To use an exposed port on the host system, we need to use either the -p flag to explicitly specify the port mapping or the -P flag, which will dynamically map the container port to the available host port. We have used the -p flag with the argument 80:80 , which will map the container port 80 to the host port 80 and make Apache accessible through the host.

The last instruction, CMD , sets the command to be executed when running the image. We are using the executable format of the CMD instruction, which specifies the executable to be run with its command-line arguments. In this case, our executable is the Apache binary with -D FOREGROUND as an argument. By default, the Apache parent process will start, create a child process, and then exit. If the Apache process exits, our container will be turned off as it no longer has a running process. With the -D FOREGROUND argument, we instruct Apache to run in the foreground and keep the parent process active. We can have only one CMD instruction in a Dockerfile.
The instruction set includes some more instructions, such as ADD , COPY , and ENTRYPOINT .

Once the image has been created, you can share it on Docker Hub, a central repository of public and private Docker images. You need an account on Docker Hub, which can be created for free. Once you get your Docker Hub credentials, you can use docker login to connect your Docker daemon with Docker Hub and then use docker push to push local images to the Docker Hub repository. You can use the respective help commands or manual pages to get more details about docker login and docker push .

Alternatively, you can also set up your own local image repository. Check out the Docker documents for deploying your own registry at https://docs.docker.com/registry/deploying/ .

We need a base image or any other image as a starting point for the Dockerfile. But how do we create our own base image?

Base images can be created with tools such as debootstrap and supermin. We need to create a distribution-specific directory structure and put all the necessary files inside it. Later, we can create a tarball of this directory structure and import the tarball as a Docker image using the docker import command.

10 Understanding Docker volumes

One of the most common questions seen on Docker forums is how to separate data from containers. This is because any data created inside containers is lost when the container gets deleted. Using docker commit to store data inside Docker images is not a good idea. To solve this problem, Docker provides an option called data volumes. Data volumes are special shared directories that can be used by one or more Docker containers. These volumes persist even when the container is deleted. These directories are created on the host file system, usually under the /var/lib/docker/ directory.

In this recipe, we will learn to use Docker volumes, share host directories with Docker containers, and learn basic backup and restore tricks that can be used with containers.

Follow these steps to understand Docker volumes:

1. To add a data volume to a container, use the -v flag with the docker run command, like so:

```
$ docker run -dP -v /var/lib/mysql --name mysql\
-e MYSQL_ROOT_PASSWORD= passwdmysql:latest
```

This will create a new MySQL container with a volume created at /var/lib/mysql inside the container. If the directory already exists on the volume path, the volume will overlay the directory contents.

2. Once the container has been started, you can get the host-specific path of the volume with the docker inspect command. Look for the Mounts section in the output of docker inspect :

```
$ docker inspect mysql
```

3. To mount a specific directory from the host system as a data volume, use the following syntax:

```
$ mkdir ~/mkdir
$ docker run -dP -v ~/mysql:/var/lib/mysql \
--name mysql mysql:latest
```

This will create a new directory named mysql at the home path and mount it as a volume inside a container at /var/lib/mysql .

4. To share a volume between multiple containers, you can use named volume containers. First, create a container with a volume attached to it. The following command will create a container with its name set to mysql :

```
$ docker run -dP -v /var/lib/mysql --name mysql\
-e MYSQL_ROOT_PASSWORD= passwd mysql:latest
```

5. Now, create a new container using the volume exposed by the mysql container and list all the files available in the container:

```
$ docker run --rm --volumes-from mysql ubuntu ls -l /var/lib/mysql
```

6. To back up data from the mysql container, use the following command:

```
$ docker run --rm--volumes-from mysql -v ~/backup:/backup \
$ tar cvf /backup/mysql.tar /var/lib/mysql
```

7. Docker volumes are not deleted when containers are removed. To delete volumes along with a container, you need to use the -v flag with the docker rm command:

```
$ dockerrm -v mysql
```

Docker volumes are designed to provide persistent storage, separate from the containers' life cycles. Even if the container gets deleted, the volume still persists unless it's explicitly specified to delete the volume with the container. Volumes can be attached while creating a container using the docker create or docker run commands. Both commands support the -v flag, which accepts volume arguments. You can add multiple volumes by repeatedly using the volume flag. Volumes can also be created in a Dockerfile using the VOLUME instruction.
When the -v flag is followed by a simple directory path, Docker creates a new directory inside

a container as a data volume. This data volume will be mapped to a directory on the host filesystem under the /var/lib/docker directory.

Docker volumes are read-write enabled by default, but you can mark a volume to be read-only using the following syntax:

```
$ docker run -dP -v /var/lib/mysql:ro --name mysql mysql:latest
```

Once a container has been created, you can get the details of all the volumes used by it, as well as its host-specific path, with the docker inspect command. The Mounts section from the output of docker inspect lists all volumes with their respective names and paths on the host system and path inside a container.

Rather than using a random location as a data volume, you can also specify a particular directory on the host to be used as a data volume. Add a host directory along with the volume argument, and Docker will map the volume to that directory:

```
$ docker run -dP -v ~/mysql:/var/lib/mysql \
--name mysql mysql:latest
```

In this case, /var/lib/mysql from the container will be mapped to the mysql directory located at the user's home address.
Need to share a single file from a host system with a container? Sure, Docker supports that too. Use docker run -v and specify the file source on the host and destination inside the container. Check out following example command:

```
$ docker run --rmd -v ~/.bash_history:/.bash_history ubuntu
```

The other option is to create a named data volume container or data-only container. You can create a named container with attached volumes and then use those volumes inside other containers using the docker run --volumes-from command. The data volumes container need not be running to access volumes attached to it. These volumes can be shared by multiple containers, plus you can create temporary, throwaway application containers by separating persistent data storage. Even if you delete a temporary container using a named volume, your data is still safe with a volume container.

From Docker version 1.9 onwards, a separate command, docker volume , is available to manage volumes. With this update, you can create and manage volumes separately from containers. Docker volumes support various backend drivers, including AUFS, OverlayFS, BtrFS, and ZFS. A simple command to create a new volume will be as follows:

```
$ docker volume create --name=myvolume
$ docker run -v myvolume:/opt alpine sh
```

11 Deploying WordPress using a Docker network

In this recipe, we will learn to use a Docker network to set up a WordPress server. We will create two containers, one for MySQL and the other for WordPress. Additionally, we will set up a private network for both MySQL and WordPress.

Let's start by creating a separate network for WordPress and the MySQL containers:

1. A new network can be created with the following command:

```
$ docker network create wpnet
```

2. Check whether the network has been created successfully with docker network ls :

```
$ docker network ls
```

3. You can get details of the new network with the docker network inspect command:

```
$ docker network inspect wpnet
```

4. Next, start a new MySQL container and set it to use wpnet :

```
$ docker run --name mysql -d \
-e MYSQL_ROOT_PASSWORD=password \
--net wpnet mysql
```

5. Now, create a container for WordPress. Make sure the WORDPRESS_DB_HOST argument matches the name given to the MySQL container:

```
$ docker run --name wordpress -d -p 80:80 \
--net wpnet\
-e WORDPRESS_DB_HOST=mysql\
-e WORDPRESS_DB_PASSWORD=password wordpress
```

6. Inspect wpnet again. This time, it should list two containers.

Now, you can access the WordPress installation at your host domain name or IP address.

Docker introduced the container networking model (CNM) with Docker version 1.9. CNM enables users to create small, private networks for a group of containers. Now, you can set up a new software-assisted network with a simple docker network create command. The Docker network supports bridge and overlay drivers for networks out of the box. You can use plugins to add other network drivers. The bridge network is a default driver used by a Docker network. It provides a network similar to the default Docker network, whereas an overlay network enables multihost networking for Docker clusters.

This recipe covers the use of a bridge network for wordpress containers. We have created a simple, isolated bridge network using the docker network command. Once the network has been created, you can set containers to use this network with the --net flag to docker run command. If your containers are already running, you can add a new network interface to them with the docker network connect command, as follows:

```
$ # docker network connect network_name container_name
$ docker network connect wpnet mysql
```

Similarly, you can use docker network disconnect to disconnect or remove a container from a specific network. Additionally, this network provides an inbuilt discovery feature. With discovery enabled, we can communicate with other containers using their names. We used this feature while connecting the MySQL container to the wordpress container. For the WORDPRESS_DB_HOST parameter, we used the container name rather than the IP address or FQDN.

If you've noticed, we have not mentioned any port mapping for the mysql container. With this new wpnet network, we need not create any port mapping on the MySQL container. The default MySQL port is exposed by the mysql container and the service is accessible only to containers running on the wpnet network. The only port available to the outside world is port 80 from the wordpress container. We can easily hide the WordPress service behind a load balancer and use multiple wordpress containers with just the load balancer exposed to the outside world.

12 Monitoring Docker containers

Docker provides inbuilt monitoring with the docker stats command, which can be used to get a live stream of the resource utilization of Docker containers.

1. To monitor multiple containers at once using their respective IDs or names, use this command:

```
$ docker stats mysql f9617f4b716c
```

2. With docker logs , you can fetch logs of your application running inside a container. This can be used similarly to the tail -f command:

```
$ docker logs -f ubuntu
```

3. Docker also records state change events from containers. These events include start, stop, create, kill, and so on. You can get real-time events with docker events :

```
$ docker events
```

To get past events, use the --since flag with docker events :

```
$ docker events --since '2015-11-01'
```

4. You can also check the changes in the container filesystem with the docker diff command. This will list newly added (A), changed (C), or deleted (D) files.

```
$ docker diff ubuntu
```

5. Another useful command is docker top , which helps look inside a container. This commands displays the processes running inside a container:

```
$ docker top ubuntu
```

13 Source

Linux: Powerful Server Administration
Uday Sawant, Oliver Pelz, Jonathan Hobson, William Leemans

Published on: April 2017
Production reference: 1130417
Published by Packt Publishing Ltd.
Livery Place

35 Livery Street
Birmingham B3 2PB, UK.
ISBN 978-1-78829-377-8