Kubernetes and Docker Bootcamp
Student Workbook
Version KD100-2.0.0

http://training.mirantis.com

Contents

Preface

Welcome

Welcome to *Kubernetes and Docker Bootcamp (KD100)*. This course covers the critical skills needed to install Docker and Kubernetes, to deploy, run, and manage containerized applications.

Course Objectives

After completing this course you will understand:

- How to install Docker
- How to use Docker to run and manage containers
- How to build Docker images
- How to configure volumes and networks in Docker
- How to install a Kubernetes cluster
- How to create Kubernetes pods, deployments, and services

This Guide Structure

This guide is designed to complement instructor-led presentations by providing step-by-step instructions for hands-on exercises.

Every chapter of this guide starts with *Chapter Details* and finishes with *Checkpoint*.

> ***Chapter Details***
>
> Each chapter opens with a short description of the chapter's goals and objectives, and an outline of the included sections.

> ***Checkpoint***
>
> Each chapter ends with a summary of what was covered in the section. You can use *Checkpoints* to verify your understanding of the topic.

There are a number of notations that are used throughout the guide. They are here to provide you with extra information on the task at hand. Do not execute the steps listed in the notations.

> ***Notes***
>
> Notes are tips, shortcuts or alternative approaches for the task at hand.

> ***Reference***
>
> References are links to external documentation relevant to a subject.

Important

Important boxes indicate a warning or caution. They detail things that are easily missed and should not be overlooked. This is information that you must be aware of before proceeding.

Code blocks:

```
Commands to be executed in a terminal window
and an example of the output
```

Commands need to be executed exactly as they are written with the exception of the command prompt and the parts enclosed in < >, which need to be substituted with your data (typically either the lab environment IP address or ID of a previously created entity).

Example of the code block:

```
stack@lab:~$ ping -c 5 <ip-address>
PING <ip-address> (<ip-address>) 56(84) bytes of data.
64 bytes from 192.168.224.2: icmp_seq=1 ttl=64 time=0.053 ms
64 bytes from 192.168.224.2: icmp_seq=2 ttl=64 time=0.052 ms
...

--- 192.168.224.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 3996ms
rtt min/avg/max/mdev = 0.045/0.051/0.053/0.005 ms
```

Here `stack@lab:~$` is the command prompt and you are expected to execute the command `ping -c 5 <ip-address>` where you should substitute `<ip-address>` by the actual IP address given in the context of that step. The actual output may be different from the ones shown in the examples. For longer outputs, the non-important part is excluded and is typically substituted with "`...`".

1. Classroom Environment

Before diving into Docker and Kubernetes, let's take some time to explore the classroom environment. Understanding how things are currently configured is beneficial for your learning as you move forward in the class.

In this lab, we will explore this currently configured environment to help with the execution of future labs.

Chapter Details	
Chapter Goal	Explore the classroom environment
Chapter Sections	*1.1. Organizational Notes* *1.2. Working with Remote Environment* *1.3. Frequently Asked Questions*

1.1. Organizational Notes

All students have their own environments to perform exercises independently. The instructor will assign personal environments at the beginning of the class.

1.2. Working with Remote Environment

For command line steps you need an SSH client:

- For Linux and OS X you should already have it.
- For Windows you can download the PuTTY SSH client from the Internet.

When working with files from command line, you can use any editors or file readers that you are familiar with and are available in default installation. Here are examples:

- Edit - `vi`, `vim`, or `nano`
- View - `less`, `more`, `cat`, or `tail`

Refer to the `man` page for specific command for details on options and usage.

Step 1 From your laptop's shell environment or using PuTTY, connect via SSH protocol to the remote environment provided by your instructor. If you did not receive information on your personal lab environment, please ask your instructor. Log in using `stack` as the user name and `b00tcamp` as the password:

```
ssh stack@<lab IP>
```

Step 2 Check that you can execute commands as `root` user:

```
stack@lab:~$ sudo su
root@lab:/home/stack#
```

Step 3 To close the `root` session press `Ctrl-D` or execute `exit`:

```
root@lab:/home/stack# exit
stack@lab:~$
```

1.3. Frequently Asked Questions

This list contains frequently asked questions that pertain to the environment or exercises.

- The specific step failed, or I cannot initiate the next step for some reason

  The environment allows you to be flexible in choosing names, IP addresses and other parameters for the steps in this course. However, the chapters in the course are connected to each other in such a way that the results from one chapter are required for the next one. We highly recommend that you use names, IP addresses and other parameters exactly as they are specified in the steps below. Also, using the same parameters will help us identify the issue. We highly recommend that you execute all the steps as listed. Please do not skip any steps or make any additional unspecified steps, such as changing the default password.

  If one specific step failed (for example, you have received unexpected error message), or you cannot initiate the next step for some reason, please check that you executed all previous steps exactly as they are specified.

  If you feel as you executed all the steps correctly, ask the instructor. Be sure to let them know of any variations you may have made to the standard flow. The instructor will help you solve the issue, and in the worst case, can revert the whole environment to the one of the predefined state, so you can continue working with the class.

Checkpoint

- Receive a lab environment assignment
- Use SSH client to access the environment
- Read Frequently Asked Questions

2. Docker Concepts

In this lab, we will install Docker and use it to run containers.

Chapter Details	
Chapter Goal	Install Docker and use it to run containers
Chapter Sections	*2.1. Install Docker* *2.2. Show running containers* *2.3. Specify a container main process* *2.4. Specify an image version* *2.5. Run an interactive container* *2.6. Run a container in a background* *2.7. Expose containers ports* *2.8. Inspect a container* *2.9. Execute a process in a container* *2.10. Copy files to/from container* *2.11. Connect a container to a network* *2.12. Restart a container* *2.13. Remove a container* *2.14. Use a Data Volume* *2.15. Use a Data Volume Container* *2.16. Link containers*

2.1. Install Docker

Step 1 We will install Docker from Docker's APT repository. For the first step, we will install HTTPS transport for Apt:

```
$ sudo apt-get update
$ sudo apt-get install -y apt-transport-https ca-certificates
```

Step 2 Add a new GPG key:

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

Step 3 Add a new Apt source entry:

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

Step 4 Update the apt package index:

```
$ sudo apt-get update
```

Step 5 Install Docker:

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

Step 6 Let's verify that Docker is installed correctly. The following command downloads a test image and runs it in a container. When the container runs, it prints an informational message and exits:

```
$ sudo docker run hello-world
Unable to find image 'hello-world:latest' locally
...
Status: Downloaded newer image for hello-world:latest

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.
 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.

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 Hub account:
 https://hub.docker.com

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/
```

Step 7 By default, the `docker` daemon binds to a Unix socket, which is owned by the user `root` and other users can access it with `sudo`. To make it possible for regular users to use the `docker` client without `sudo` there is a group called `docker`. When the `docker` daemon starts, it makes the ownership of the socket read/writable by the `docker` group.

Let's add the current user `stack` to the `docker` group:

```
$ sudo usermod -aG docker stack
```

Step 8 Usually, to make the changes to take affect, you need to logout and login again, but in our case, we will create a new login session:

```
$ sudo su - stack
```

Step 9 Check that the user `stack` is in the `docker` group:

```
$ id
uid=1000(stack) gid=1000(stack) groups=1000(stack),999(docker)
```

Step 10 Now you can use `docker` without `sudo`:

```
$ docker run hello-world

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

2.2. Show running containers

Step 1 Run `docker ps` to show running containers:

```
$ docker ps
CONTAINER ID IMAGE       COMMAND  CREATED        STATUS                      PORTS NAMES
```

Step 2 The output shows that there are no running containers at the moment. Use the command `docker ps -a` to list all containers:

```
docker ps -a
CONTAINER ID IMAGE       COMMAND  CREATED        STATUS                     PORTS  NAMES
6e6db2a24a8e hello-world "/hello" 15 minutes ago Exited (0) 15 minutes ago         dreamy_nobel
77609b91727a hello-world "/hello" 10 minutes ago Exited (0) 10 minutes ago         grave_pike
```

In the previous section we started two containers and the command `docker ps -a` shows that both of them are exited. Note that Docker has generated random names for the containers (the last column). In your case, these names can be different.

Step 3 Let's run the command `docker images` to show all the images on your local system:

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
hello-world         latest              c54a2cc56cbb        4 weeks ago         1.848 kB
```

As you see, there is only one image that was downloaded from the Docker Hub.

2.3. Specify a container main process

Step 1 Let's run our own "hello world" container. We will use the existing Ubuntu image for that:

```
$ docker run ubuntu /bin/echo 'Hello world'
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
...
Status: Downloaded newer image for ubuntu:latest
Hello world
```

As you see, Docker downloaded the image `ubuntu` because it was not on the local machine.

Step 2 Let's run the command `docker images` again:

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              latest              42118e3df429        11 days ago         124.8 MB
hello-world         latest              c54a2cc56cbb        4 weeks ago         1.848 kB
```

Step 3 If you run the same "hello world" container again, Docker will use a local copy of the image:

```
$ docker run ubuntu /bin/echo 'Hello world'
Hello world
```

2.4. Specify an image version

Step 1 As you see, Docker has downloaded the `ubuntu:latest` image. You can see Ubuntu version by running the following command:

```
$ docker run ubuntu /bin/cat /etc/issue.net
Ubuntu 16.04 LTS
```

Let's say you need a previous Ubuntu LTS release. In this case, you can specify the version you need:

```
$ docker run ubuntu:14.04 /bin/cat /etc/issue.net
Unable to find image 'ubuntu:14.04' locally
14.04: Pulling from library/ubuntu
...
Status: Downloaded newer image for ubuntu:14.04
Ubuntu 14.04.4 LTS
```

Step 2 The `docker images` command should show that we have two Ubuntu images downloaded locally:

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              latest              42118e3df429        11 days ago         124.8 MB
ubuntu              14.04               0ccb13bf1954        11 days ago         188 MB
hello-world         latest              c54a2cc56cbb        4 weeks ago         1.848 kB
```

2.5. Run an interactive container

Step 1 Let's use the ubuntu image to run an interactive bash session. We will use the `-t` command line argument to assigns a pseudo-tty or terminal inside the new container, and the `-i` to make an interactive connection by grabbing the standard input of the container. Use the `exit` command or press `Ctrl-D` to exit the interactive bash session:

```
$ docker run -t -i ubuntu /bin/bash
root@17d8bdeda98e:/# uname -a
Linux 17d8bdeda98e 3.19.0-31-generic ...
root@17d8bdeda98e:/# exit
exit
```

Step 2 Let's check that when the shell process has finished, the container stops:

```
$ docker ps
CONTAINER ID IMAGE       COMMAND  CREATED        STATUS                      PORTS NAMES
```

2.6. Run a container in a background

Step 1 To run a container in a background use the `-d` command line argument:

```
$ docker run -d ubuntu /bin/sh -c "while true; do echo hello world; sleep 1; done"
ac231579e57faf1afcb76e4f7ff22415d39af73eeeb4476eab3ea6bb8991f556
```

The command returned the container ID (it can be different in your case).

Step 2 Let's use the `docker ps` command to see running containers:

```
$ docker ps
CONTAINER ID IMAGE  COMMAND                  CREATED        STATUS        PORTS NAMES
ac231579e57f ubuntu "/bin/sh -c 'while tr"   1 minute ago   Up 11 minute        evil_golick
```

`ac231579e57f` is the shortened container ID (it can be different in your case).

Step 3 Let's use this short container ID to show the container standard output:

```
$ docker logs <short-id>
hello world
hello world
hello world
...
```

As we see, in the `docker ps` command output, the auto generated container name is `evil_golick` (your container can have a different name).

Step 4 Use your container name to to show the container standard output:

```
$ docker logs <name>
hello world
hello world
hello world
...
```

Step 5 Finally, let's stop our container:

```
$ docker stop <name>
<name>
```

Step 6 Check, that there are no running containers:

```
$ docker ps
CONTAINER ID IMAGE       COMMAND  CREATED        STATUS                      PORTS NAMES
```

2.7. Expose containers ports

Step 1 Let's run a simple web application. We will use the existing image `training/webapp`, which contains a Python Flask application:

```
$ docker run -d -P training/webapp python app.py
...
Status: Downloaded newer image for training/webapp:latest
6e88f42d3d853762edcbfe1fe73fdc5c48865275bc6df759b83b0939d5bd2456
```

In the command above we specified the main process (`python app.py`), the `-d` command line argument, which tells Docker to run the container in the background. The `-P` command line argument tells Docker to map any required network ports inside our container to our host. This allows us to access the web application in the container.

Step 2 Use the `docker ps` command to list running containers:

```
$ docker ps
CONTAINER ID IMAGE           COMMAND         CREATED       STATUS       PORTS                     NAMES
6e88f42d3d85 training/webapp "python app.py" 3 minutes ago Up 3 minutes 0.0.0.0:32768->5000/tcp determined_torvalds
```

The `PORTS` column contains the mapped ports. In our case, Docker has exposed port 5000 (the default Python Flask port) on port 32768 (can be different in your case).

Step 3 The `docker port` command shows the exposed port. We will use the container name (`determined_torvalds` in the example above, it can be different in your case):

```
$ docker port <name> 5000
0.0.0.0:32768
```

Step 4 Let's check that we can access the web application exposed port:

```
$ curl http://localhost:<port>/
Hello world!
```

Step 5 Let's stop our web application for now:

```
$ docker stop <name>
```

Step 6 We want to manually specify the local port to expose (`-p` argument). Let's use the standard HTTP port 80. We also want to specify the container name (`--name` argument):

```
$ docker run -d -p 80:5000 --name webapp training/webapp python app.py
249476631f7d445c6a61a6067c4c24461b0e51ea0667c86cb1bce301981ecd22
```

Step 7 Let's check that the port 80 is exposed:

```
$ docker ps
CONTAINER ID IMAGE           COMMAND         CREATED       STATUS       PORTS                NAMES
249476631f7d training/webapp "python app.py" 1 minute  ago Up 1 minute  0.0.0.0:80->5000/tcp webapp

$ curl http://localhost/
Hello world!
```

2.8. Inspect a container

Step 1 You can use the `docker inspect` command to see the configuration and status information for the specified container:

```
$ docker inspect webapp
[
```

```
    {
        "Id": "249476631f7d...",
        "Created": "2016-08-02T23:42:56.932135327Z",
        "Path": "python",
        "Args": [
            "app.py"
        ],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 16055,
            "ExitCode": 0,
            "Error": "",
            ...
```

Step 2 You can specify a filter (`-f` command line argument) to show only specific elements. For example:

```
$ docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' webapp
172.17.0.2
```

This command returned the IP address of the container.

2.9. Execute a process in a container

Step 1 Use the `docker exec` command to execute a command in the running container. For example:

```
$ docker exec webapp ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.2  0.0  52320 17384 ?        Ss   00:11   0:00 python app.py
root        26  0.0  0.0  15572  2104 ?        Rs   00:12   0:00 ps aux
```

The same command with the `-it` command line argument can be used to run an interactive session in the container:

```
$ docker exec -it webapp bash
root@249476631f7d:/opt/webapp# ps auxw
ps auxw
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0  52320 17384 ?        Ss   00:11   0:00 python app.py
root        32  0.0  0.0  18144  3064 ?        Ss   00:14   0:00 bash
root        47  0.0  0.0  15572  2076 ?        R+   00:16   0:00 ps auxw
```

Step 2 Use the `exit` command or press `Ctrl-D` to exit the interactive bash session:

```
root@249476631f7d:/opt/webapp# exit
```

2.10. Copy files to/from container

The `docker cp` command allows you to copy files from the container to the local machine or from the local file system to the container. This command works for a running or stopped container.

Step 1 Let's copy the container's `app.py` file to the local machine:

```
$ docker cp webapp:/opt/webapp/app.py .
```

Step 2 Edit the local `app.py` file. For example, change the line `return 'Hello '+provider+'!'` to `return 'Hello '+provider+'!!!'`. Copy the modified file back and restart the container:

```
$ docker cp app.py webapp:/opt/webapp/
$ docker restart webapp
```

Step 3 Check that the modified web application works::

```
$ curl http://localhost/
Hello world!!!
```

2.11. Connect a container to a network

Step 1 Start a database container:

```
$ docker run -d --name db training/postgres
```

Step 2 Check that the `webapp` and `db` containers are running:

```
$ docker ps
CONTAINER ID IMAGE             COMMAND                CREATED        STATUS        PORTS                NAMES
...
c3afff20019a training/postgres "su postgres -c '/usr" 4 minutes ago  Up 4 minutes  5432/tcp             db
62ed4a627356 training/webapp   "python app.py"        30 minutes ago Up 30 minutes 0.0.0.0:80->5000/tcp webapp
```

Step 3 Get IP addresses of the `webapp` and `db` containers:

```
$ docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' webapp
172.17.0.3

$ docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' db
172.17.0.4
```

Step 4 In your case, the IP addresses can be different. Let's start an interactive session in the `db` container and ping the IP address of the `webapp`:

```
$ docker exec -it db bash
root@c3afff20019a:/# ping -c 1 172.17.0.3
PING 172.17.0.3 (172.17.0.3) 56(84) bytes of data.
64 bytes from 172.17.0.3: icmp_seq=1 ttl=64 time=0.111 ms

--- 172.17.0.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.111/0.111/0.111/0.000 ms
root@c3afff20019a:/# exit
```

Step 5 The `docker network ls` command shows Docker networks:

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
d428e49e4869        bridge              bridge              local
0d1f78528cc5        host                host                local
4a07cef84617        none                null                local
```

Step 6 You can inspect your containers to see the networks they are connected to:

```
$ docker inspect -f '{{json .NetworkSettings.Networks}}' webapp
{"bridge": ...


$ docker inspect -f '{{json .NetworkSettings.Networks}}' db
{"bridge": ...
```

Step 7 You can inspect the `bridge` network to get IP address of all containers in this network:

```
$ docker network inspect bridge
...
"Containers": {
    "62ed4a627356...": {
        "Name": "webapp",
        "EndpointID": "7af40b8967c0...",
        "MacAddress": "02:42:ac:11:00:03",
        "IPv4Address": "172.17.0.3/16",
        "IPv6Address": ""
    },
    "c3afff20019a...": {
        "Name": "db",
        "EndpointID": "1458900b4ebc...",
        "MacAddress": "02:42:ac:11:00:04",
        "IPv4Address": "172.17.0.4/16",
        "IPv6Address": ""
    }
},
...
```

Step 8 By default, Docker runs containers in the `bridge` network. You may want to isolate one or more containers in a separate network. Let's create a new network:

```
$ docker network create my-network \
-d bridge \
--subnet 172.19.0.0/16
```

The `-d bridge` command line argument specifies the `bridge` network driver and the `--subnet` command line argument specifies the network segment in CIDR format. If you do not specify a subnet when creating a network, then Docker assigns a subnet automatically, so it is a good idea to specify a subnet to avoid potential conflicts with the existing networks.

Step 9 To check that the new network is created, execute `docker network ls`:

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
```

```
d428e49e4869        bridge              bridge              local
0d1f78528cc5        host                host                local
56ef0481820d        my-network          bridge              local
4a07cef84617        none                null                local
```

Step 10 Let's inspect the new network:

```
$ docker network inspect my-network
[
    {
        "Name": "my-network",
        "Id": "56ef0481820d...",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.19.0.0/16"
                }
            ]
        },
        "Internal": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```

Step 11 As expected, there are no containers connected to the `my-network`. Let's recreate the `db` container in the `my-network`:

```
$ docker rm -f db

$ docker run -d --network=my-network --name db training/postgres
```

Step 12 Inspect the `my-network` again:

```
$ docker network inspect my-network
...
    "Containers": {
        "93af62cdab64...": {
            "Name": "db",
            "EndpointID": "b1e8e314cff0...",
            "MacAddress": "02:42:ac:12:00:02",
            "IPv4Address": "172.19.0.2/16",
            "IPv6Address": ""
        }
    },
```

```
...
```

As you see, the `db` container is connected to the `my-network` and has `172.19.0.2` address.

Step 13 Let's start an interactive session in the `db` container and ping the IP address of the `webapp` again:

```
$ docker exec -it db bash
root@c3afff20019a:/# ping -c 1 172.17.0.3
PING 172.17.0.3 (172.17.0.3) 56(84) bytes of data.

--- 172.17.0.3 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

As expected, the `webapp` container is no longer accessible from the `db` container, because they are connected to different networks.

Step 14 Let's connect the `webapp` container to the `my-network`:

```
$ docker network connect my-network webapp
```

Step 15 Check that the `webapp` container now is connected to the `my-network`:

```
$ docker network inspect my-network
...
    "Containers": {
        "62ed4a627356...": {
            "Name": "webapp",
            "EndpointID": "ae95b0103bbc...",
            "MacAddress": "02:42:ac:12:00:03",
            "IPv4Address": "172.19.0.3/16",
            "IPv6Address": ""
        },
        "93af62cdab64...": {
            "Name": "db",
            "EndpointID": "b1e8e314cff0...",
            "MacAddress": "02:42:ac:12:00:02",
            "IPv4Address": "172.19.0.2/16",
            "IPv6Address": ""
        }
    },
...
```

The output shows that two containers are connected to the `my-network` and the `webapp` container has `172.19.0.3` address in that network.

Step 16 Check that the `webapp` container is accessible from the `db` container using its new IP address:

```
$ docker exec -it db bash
root@c3afff20019a:/# ping -c 1 172.19.0.3
PING 172.19.0.3 (172.19.0.3) 56(84) bytes of data.
64 bytes from 172.19.0.3: icmp_seq=1 ttl=64 time=0.136 ms
```

```
--- 172.19.0.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.136/0.136/0.136/0.000 ms
```

2.12. Restart a container

Step 1 Let's stop the container with web application:

```
$ docker stop webapp
```

The main process inside the container will receive `SIGTERM`, and after a grace period, `SIGKILL`.

Step 2 You can start the container later using the `docker start` command:

```
$ docker start webapp
```

Step 3 Check that the web application works:

```
$ curl http://localhost/
Hello world!
```

Step 4 You also can restart the running container using the `docker restart` command. The `-t` command line argument specifies a number of seconds to wait for stop before killing the container:

```
$ docker restart webapp
```

2.13. Remove a container

Step 1 You can remove the existing container. You should stop the container before removing it. Alternatively you can use the `-f` command line argument:

```
$ docker rm -f webapp
$ docker rm -f db
```

2.14. Use a Data Volume

Step 1 Add a data volume to a container:

```
$ docker run -d -P --name webapp -v /webapp training/webapp python app.py
```

This command started a new container and created a new volume inside the container at `/webapp`.

Step 2 Locate the volume on the host using the `docker inspect` command:

```
$ docker inspect webapp
...
    "Mounts": [
        {
            "Name": "7d72348e8f...",
            "Source": "/var/lib/docker/volumes/7d72348e8f.../_data",
            "Destination": "/webapp",
            "Driver": "local",
            "Mode": "",
            "RW": true,
```

```
            "Propagation": ""
        }
    ],
...
```

Alternatively, you can specify a host directory you want to use as a data volume:

```
$ mkdir db
$ docker run -d --name db -v /home/stack/db:/db training/postgres
```

Step 3 Start an interactive session in the `db` container and create a new file in the `/db` directory:

```
$ docker exec -it db bash
root@9a7a4fbcc929:/# cd /db
root@9a7a4fbcc929:/db# touch hello_from_db_container
root@9a7a4fbcc929:/db# exit
```

Step 4 Check that the local `db` directory contains the new file:

```
$ ls db
hello_from_db_container
```

Step 5 Check that the data volume is persistent. Remove the `db` container:

```
$ docker rm -f db
```

Step 6 Create the `db` container again:

```
$ docker run -d --name db -v /home/stack/db:/db training/postgres
```

Step 7 Check that its `/db` directory contains the `hello_from_db_container` file:

```
$ docker exec -it db bash
root@47a60c01590e:/# ls /db
hello_from_db_container
root@47a60c01590e:/# exit
```

2.15. Use a Data Volume Container

Step 1 Let's create a new named container with a volume to share:

```
$ docker create -v /dbdata --name dbstore training/postgres /bin/true
```

This container does not run an application and reuses the `training/postgres` image.

Step 2 Use the `--volumes-from` flag to mount the `/dbdata` volume in another containers:

```
$ docker run -d --volumes-from dbstore --name db1 training/postgres
$ docker run -d --volumes-from dbstore --name db2 training/postgres
```

2.16. Link containers

Step 1 You should have the `db` and `webapp` containers running. Remove the `webapp` container:

```
$ docker rm -f webapp
```

Step 2 Create a new `webapp` container and link it with your `db` container:

```
$ docker run -d -P --name webapp --link db:db training/webapp python app.py
```

Step 3 Inspect your linked containers with `docker inspect`:

```
$ docker inspect -f "{{ .HostConfig.Links }}" webapp
[/db:/webapp/db]
```

You see that the `webapp` container is now linked to the `db` container. This allows it to access information about the `db` container.

Step 4 Start an interactive session in the `webapp` container and check its `/etc/hosts` file and its environment variables:

```
$ docker exec -it webapp bash
root@90eb1de6fa8a:/opt/webapp# env | grep DB
DB_NAME=/webapp/db
DB_PORT_5432_TCP_ADDR=172.17.0.3
DB_PORT=tcp://172.17.0.3:5432
DB_PORT_5432_TCP=tcp://172.17.0.3:5432
DB_PORT_5432_TCP_PORT=5432
DB_PORT_5432_TCP_PROTO=tcp
DB_ENV_PG_VERSION=9.3
root@90eb1de6fa8a:/opt/webapp# cat /etc/hosts
...
172.17.0.3  db 47a60c01590e
172.17.0.2  90eb1de6fa8a
root@90eb1de6fa8a:/opt/webapp# exit
```

You see that Docker updated the `/etc/hosts` file and set the environment variables. This information can be used in the `webapp` container to access the database.

Checkpoint

- Install Docker
- List running containers
- Specify a container main process
- Run an interactive container
- Run a container in a background
- Use a container ID and container name to display container standard output
- Expose container ports
- Execute a process in a container
- Copy files to/from container
- Connect a container to a network
- Use a data volume
- Use a data volume container
- Link containers
- Stop a container
- Remove a container

3. Docker Images

In this lab, we will write a Dockerfile and build a new Docker image.

Chapter Details	
Chapter Goal	Learn how to write Dockerfile and build Docker image
Chapter Sections	*3.1. Build a Docker image* *3.2. Build a multi-container application*

3.1. Build a Docker image

In the previous section (*2. Docker Concepts*), we learned how to use Docker images to run Docker containers. Docker images that we used have been downloaded from the Docker Hub, a registry of Docker images. In this section we will create a simple web application from scratch. We will use Flask (http://flask.pocoo.org/), a microframework for Python. Our application for each request will display a random picture from the defined set.

In the next session we will create all necessary files for our application. The code for this application is also available in GitHub:

https://github.com/docker/labs/tree/master/beginner/flask-app

3.1.1. Create project files

Step 1 Create a new directory `flask-app` for our project:

```
$ mkdir flask-app
$ cd flask-app
```

Step 2 In this directory, we will create the following project files:

```
flask-app/
    Dockerfile
    app.py
    requirements.txt
    templates/
        index.html
```

Step 3 Create a new file `app.py` with the following content:

```
from flask import Flask, render_template
import random

app = Flask(__name__)

# list of cat images
images = [
    "http://ak-hdl.buzzfed.com/static/2013-10/enhanced/webdr05/15/9/anigif_enhanced-buzz-26388-1381844103-11.gif",
    "http://ak-hdl.buzzfed.com/static/2013-10/enhanced/webdr01/15/9/anigif_enhanced-buzz-31540-1381844535-8.gif",
    "http://ak-hdl.buzzfed.com/static/2013-10/enhanced/webdr05/15/9/anigif_enhanced-buzz-26390-1381844163-18.gif",
    "http://ak-hdl.buzzfed.com/static/2013-10/enhanced/webdr06/15/10/anigif_enhanced-buzz-1376-1381846217-0.gif",
    "http://ak-hdl.buzzfed.com/static/2013-10/enhanced/webdr03/15/9/anigif_enhanced-buzz-3391-1381844336-26.gif",
    "http://ak-hdl.buzzfed.com/static/2013-10/enhanced/webdr06/15/10/anigif_enhanced-buzz-29111-1381845968-0.gif",
    "http://ak-hdl.buzzfed.com/static/2013-10/enhanced/webdr03/15/9/anigif_enhanced-buzz-3409-1381844582-13.gif",
    "http://ak-hdl.buzzfed.com/static/2013-10/enhanced/webdr02/15/9/anigif_enhanced-buzz-19667-1381844937-10.gif",
    "http://ak-hdl.buzzfed.com/static/2013-10/enhanced/webdr05/15/9/anigif_enhanced-buzz-26358-1381845043-13.gif",
    "http://ak-hdl.buzzfed.com/static/2013-10/enhanced/webdr06/15/9/anigif_enhanced-buzz-18774-1381844645-6.gif",
    "http://ak-hdl.buzzfed.com/static/2013-10/enhanced/webdr06/15/9/anigif_enhanced-buzz-25158-1381844793-0.gif",
    "http://ak-hdl.buzzfed.com/static/2013-10/enhanced/webdr03/15/10/anigif_enhanced-buzz-11980-1381846269-1.gif"
]

@app.route('/')
def index():
    url = random.choice(images)
    return render_template('index.html', url=url)

if __name__ == "__main__":
    app.run(host="0.0.0.0")
```

Step 4 Create a new file `requirements.txt` with the following line:

```
Flask==0.10.1
```

Step 5 Create a new directory `templates` and create a new file `index.html` in this directory with the following content:

```
<html>
  <head>
    <style type="text/css">
      body {
        background: black;
        color: white;
      }
      div.container {
        max-width: 500px;
        margin: 100px auto;
        border: 20px solid white;
        padding: 10px;
        text-align: center;
      }
      h4 {
        text-transform: uppercase;
      }
    </style>
  </head>
  <body>
    <div class="container">
      <h4>Cat Gif of the day</h4>
      <img src="{{url}}" />
    </div>
  </body>
</html>
```

Step 6 Create a new file `Dockerfile`:

```
# our base image
FROM alpine:latest

# Install python and pip
RUN apk add --update py-pip

# upgrade pip
RUN pip install --upgrade pip

# install Python modules needed by the Python app
COPY requirements.txt /usr/src/app/
RUN pip install --no-cache-dir -r /usr/src/app/requirements.txt

# copy files required for the app to run
COPY app.py /usr/src/app/
COPY templates/index.html /usr/src/app/templates/

# tell the port number the container should expose
EXPOSE 5000
```

```
# run the application
CMD ["python", "/usr/src/app/app.py"]
```

3.1.2. Build a Docker image

Step 1 Now we can build our Docker image. In the command below, replace `<user-name>` with your user name. For a real image, the user name should be the same as you created when you registered on Docker Hub. Because we will not publish our image, you can use any valid name:

```
$ docker build -t <user-name>/myfirstapp .
Sending build context to Docker daemon 8.192 kB
Step 1 : FROM alpine:latest
...
Successfully built f1277efd5a79
```

Step 2 Check that your image exists:

```
$ docker images
REPOSITORY     TAG    IMAGE ID     CREATED       SIZE
.../myfirstapp latest f1277efd5a79 6 minutes ago 56.34 MB
```

Step 3 Run a container in a background and expose a standart HTTP port (80), which is redirected to the container's port 5000:

```
$ docker run -dp 80:5000 --name myfirstapp <user-name>/myfirstapp
```

Step 4 Use your browser to open the address `http://<lab IP>` and check that the application works.

Step 5 Stop the container and remove it:

```
$ docker stop myfirstapp
myfirstapp
$ docker rm myfirstapp
myfirstapp
```

3.2. Build a multi-container application

In this section, will guide you through the process of building a multi-container application. The application code is available at GitHub:

https://github.com/docker/example-voting-app

Step 1 Clone the existing application from GitHub:

```
$ git clone https://github.com/docker/example-voting-app.git
Cloning into 'example-voting-app'...
...

$ cd example-voting-app
```

Step 2 Edit the `vote/app.py` file, change the following lines near the top of the file:

```
option_a = os.getenv('OPTION_A', "Cats")
option_b = os.getenv('OPTION_B', "Dogs")
```

Step 3 Replace "Cats" and "Dogs" with two options of your choice. For example:

```
option_a = os.getenv('OPTION_A', "Java")
option_b = os.getenv('OPTION_B', "Python")
```

Step 4 The existing file `docker-compose.yml` defines several images:

- A voting-app container based on a Python image
- A result-app container based on a Node.js image
- A Redis container based on a redis image, to temporarily store the data.
- A worker app based on a dotnet image
- A Postgres container based on a postgres image

Note that three of the containers are built from Dockerfiles, while the other two are images on Docker Hub.

Step 5 Let's change the default port to expose. Edit the `docker-compose.yml` file and find the following lines:

```
...
ports:
  - "5000:80"
...
```

Change *5000* to *80*:

```
...
ports:
  - "80:80"
...
```

Step 6 Install the `docker-compose` tool:

```
$ curl -Lo docker-compose \
https://github.com/docker/compose/releases/download/1.8.0/docker-compose-Linux-x86_64 \
&& chmod +x docker-compose \
&& sudo mv docker-compose /usr/local/bin/
```

Step 7 Use the `docker-compose` tool to launch your application:

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

Step 8 Check that all containers are running:

```
$ docker ps | grep examplevotingapp
59e355151f56 examplevotingapp_worker "/bin/sh -c 'dotnet W" 2 minutes ago Up 2 minutes                                          examplevotingapp_worker_1
fd0740ee1525 redis:alpine            "docker-entrypoint.sh" 2 minutes ago Up 2 minutes      0.0.0.0:32772->6379/tcp               examplevotingapp_redis_1
1fd3b378dd0a examplevotingapp_result "nodemon --debug serv" 2 minutes ago Up 2 minutes      0.0.0.0:5858->5858/tcp, 0.0.0.0:5001->80/tcp   examplevotingapp_result_1
0948fd9ba26c postgres:9.4            "/docker-entrypoint.s" 2 minutes ago Up 2 minutes      5432/tcp                              examplevotingapp_db_1
52ef44aa1b97 examplevotingapp_vote   "python app.py"        2 minutes ago Up About a minute 0.0.0.0:80->80/tcp                    examplevotingapp_vote_1
```

Step 9 Use your browser to open the address `http://<lab IP>` and check that the application works. Then stop the application:

```
$ docker-compose down
Stopping examplevotingapp_worker_1 ... done
Stopping examplevotingapp_redis_1 ... done
```

```
Stopping examplevotingapp_result_1 ... done
Stopping examplevotingapp_db_1 ... done
Stopping examplevotingapp_vote_1 ... done
Removing examplevotingapp_worker_1 ... done
Removing examplevotingapp_redis_1 ... done
Removing examplevotingapp_result_1 ... done
Removing examplevotingapp_db_1 ... done
Removing examplevotingapp_vote_1 ... done
Removing network examplevotingapp_default
```

Checkpoint

- Write Dockerfile
- Build an image
- Install Docker Compose
- Write docker-compose.yml file
- Use Docker Compose to build and run a multi-container application

4. Kubernetes Concepts

In this lab, we will install a single-node Kubernetes cluster and explore it basic features.

Chapter Details	
Chapter Goal	Install and use a single-node Kubernetes cluster
Chapter Sections	*4.1. Install a single-node Kubernetes cluster* *4.2. Kubernetes client* *4.3. Create a Pod* *4.4. Create a Replication Controller* *4.5. Create a Service* *4.6. Delete a Service, Controller, Pod* *4.7. Create a Deployment* *4.8. Create a Job*

4.1. Install a single-node Kubernetes cluster

Minikube (https://github.com/kubernetes/minikube) is a tool that allows running a single-node Kubernetes cluster in a virtual machine. It can be used on GNU/Linux os OS X and requires VirtualBox, KVM (for Linux), xhyve (OS X), or VMware Fusion (OS X) to be installed on your computer. Minikube creates a new virtual machine with GNU/Linux, installs and configures Docker and Kubernetes, runs a Kubernetes cluster. You can use Minikube on your laptop to explore Kubernetes features.

In this lab, we will install a single-node Kubernetes cluster locally using `kube-deploy`.

Step 1 Install the kubectl command line tool:

```
$ curl -Lo kubectl \
http://storage.googleapis.com/kubernetes-release/release/v1.4.1/bin/linux/amd64/kubectl \
&& chmod +x kubectl \
&& sudo mv kubectl /usr/local/bin/
```

Step 2 Clone the kube-deploy project from GitHub:

```
$ git clone https://github.com/kubernetes/kube-deploy
```

Step 3 Start a local Kubernetes cluster:

```
$ cd kube-deploy/docker-multinode
$ sudo ./master.sh
...
Done. It may take about a minute before apiserver is up.
```

The local Kubernetes cluster is up and running. Check that kubectl can connect to the Kubernetes cluster:

```
$ kubectl version
Client Version: version.Info{Major:"1", Minor:"4", ...
Server Version: version.Info{Major:"1", Minor:"4", ...
```

4.2. Kubernetes client

The kubectl tools allows connecting to multiple Kubernetes clusters. A set of parameters, for example, the address the Kubernetes API server, credentials is called a context. You can define several contexts and specify which context to use to connect to a specific cluster. Also you can specify a default context to use.

Step 1 Use `kubectl config view` to view the current kubectl configuration:

```
$ kubectl config view
apiVersion: v1
clusters: []
contexts: []
current-context: ""
kind: Config
preferences: {}
users: []
```

As you see there is no context defined. In this case, kubectl uses `http://localhost:8080` as a default address of the Kubernetes API server.

Step 2 Specify the address of the Kubernetes API server via kubectl command line:

```
$ kubectl -s http://localhost:8080 version
Client Version: version.Info{Major:"1", Minor:"4", ...
Server Version: version.Info{Major:"1", Minor:"4", ...
```

Step 3 Let's define a new context, set the address of the Kubernetes API server, and make this context the default one:

```
$ kubectl config set-cluster default --server=http://localhost:8080
cluster "default" set.

$ kubectl config set-context default --cluster=default
context "default" set.

$ kubectl config use-context default
switched to context "default".
```

Step 4 Use `kubectl config view` to view the current kubectl configuration:

```
$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    server: http://localhost:8080
  name: default
contexts:
- context:
    cluster: default
    user: ""
  name: default
current-context: default
kind: Config
preferences: {}
users: []
```

4.3. Create a Pod

We are going to begin declaring Kubernetes resources by writing YAML files containing resource definitions. To make our lives easier, let's create a simple .vimrc configuration file that will deal with indentation and whitespaces the way we want.

Step 1 Create vim configuration file at `/home/stack/.vimrc` and populate it with the contents below:

```
set expandtab
set tabstop=2
```

Step 2 Define a new pod in the file `echoserver-pod.yaml`:

```
apiVersion: v1
kind: Pod
metadata:
```

```
  name: echoserver
spec:
  containers:
  - name: echoserver
    image: gcr.io/google_containers/echoserver:1.4
    ports:
    - containerPort: 8080
```

We use the existing image `echoserver`. This is a simple server that responds with the http headers it received. It runs on nginx server and implemented using lua in the nginx configuration: https://github.com/kubernetes/contrib/tree/master/ingress/echoheaders

Step 3 Create the `echoserver` pod:

```
$ kubectl create -f echoserver-pod.yaml
pod "echoserver" created
```

Step 4 Use `kubectl get pods` to watch the pod get created:

```
$ kubectl get pods
NAME         READY     STATUS    RESTARTS   AGE
echoserver   1/1       Running   0          5s
```

Step 5 Now let's get the pod definition back from Kubernetes:

```
$ kubectl get pods echoserver -o yaml > echoserver-pod-created.yaml
```

Compare `echoserver-pod.yaml` and `echoserver-pod-created.yaml` to see additional properties that have been added to the original pod definition.

4.4. Create a Replication Controller

Step 1 Define a new replication controller for 2 replicas for the pod `echoserver`. Create a new file `echoserver-rc.yaml` with the following content:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: echoserver
spec:
  replicas: 2
  selector:
    app: echoserver
  template:
    metadata:
      name: echoserver
      labels:
        app: echoserver
    spec:
      containers:
      - name: echoserver
        image: gcr.io/google_containers/echoserver:1.4
        ports:
```

```
        - containerPort: 8080
```

Step 2 Create a new replication controller:

```
$ kubectl create -f echoserver-rc.yaml
replicationcontroller "echoserver" created
```

Step 3 Use `kubectl get replicationcontrollers` to list replication controllers:

```
$ kubectl get replicationcontrollers
NAME         DESIRED   CURRENT   AGE
echoserver   2         2         1m
```

Step 4 Use `kubectl get pods` to list pods:

```
$ kubectl get pods
NAME               READY     STATUS    RESTARTS   AGE
echoserver         1/1       Running   0          37m
echoserver-obzuw   1/1       Running   0          46s
echoserver-rl8kx   1/1       Running   0          46s
```

Step 5 Our replication controller created two new pods (replicas). The existing pod `echoserver` does not have the label `app: echoserver`, therefore it is not controlled by our replication controller. Let's add this label the `echoserver` pod:

```
$ kubectl label pods echoserver app=echoserver
pod "echoserver" labeled
```

Step 6 List pods:

```
$ kubectl get pods
NAME               READY     STATUS    RESTARTS   AGE
echoserver         1/1       Running   0          1h
echoserver-rl8kx   1/1       Running   0          1h
```

Step 7 Our replication controller has detected that there are three pods labeled with `app: echoserver`, so one pod has been stopped by the controller. Use `kubectl describe` to see controller events:

```
$ kubectl describe replicationcontroller/echoserver
Name:               echoserver
Namespace:  default
Image(s):   gcr.io/google_containers/echoserver:1.4
Selector:   app=echoserver
Labels:             app=echoserver
Replicas:   2 current / 2 desired
Pods Status:        2 Running / 0 Waiting / 0 Succeeded / 0 Failed
No volumes.
Events:
  ... Reason                        Message
  ... --------                      -------
  ... SuccessfulDelete      Deleted pod: echoserver-obzuw
```

Step 8 To scale the number of replicas up we need to update the field `replicas`. Edit the file `echoserver-rc.yaml`, change the number of replicas to 3. Then use `kubectl replace` to update the replication controller:

```
$ kubectl replace -f echoserver-rc.yaml
replicationcontroller "echoserver" replaced
```

Step 9 Use `kubectl describe` to check that the number of replicas has been updated in the controller:

```
$ kubectl describe replicationcontroller/echoserver
...
Replicas:   3 current / 3 desired
Pods Status:        3 Running / 0 Waiting / 0 Succeeded / 0 Failed
...
```

Step 10 Let's check the number of pods:

```
$ kubectl get pods
NAME               READY     STATUS    RESTARTS   AGE
echoserver         1/1       Running   0          2h
echoserver-b5ujn   1/1       Running   0          2m
echoserver-rl8kx   1/1       Running   0          1h
```

You can see that the replication controller has started a new pod.

4.5. Create a Service

We have three running `echoserver` pods, but we cannot use them from our lab, because the container ports are not accessible. Let's define a new service that will expose `echoserver` ports and make them accessible from the lab.

Step 1 Create a new file `echoserver-service.yaml` with the following content:

```
apiVersion: v1
kind: Service
metadata:
  name: echoserver
spec:
  type: "NodePort"
  ports:
    - port: 8080
      protocol: TCP
      targetPort: 8080
  selector:
    app: echoserver
```

Step 2 Create a new service:

```
$ kubectl create -f echoserver-service.yaml
service "echoserver" created
```

Step 3 Check the service details:

```
$ kubectl describe services/echoserver
Name:                echoserver
Namespace:           default
Labels:              <none>
Selector:            app=echoserver
Type:                NodePort
IP:                  ...
Port:                <unset> 8080/TCP
NodePort:            <unset> 31698/TCP
Endpoints:           ...:8080,...:8080,..:8080
Session Affinity:    None
No events.
```

Note that the output contains three endpoints and a node port. It is 31698 in the output above, but it can be different in your case. Remember this port to use it in the next step.

Step 4 To access a service exposed via a node port, specify the node port from the previous step:

```
$ curl http://localhost:<port>
CLIENT VALUES:
client_address=...
command=GET
real path=/
...
```

4.6. Delete a Service, Controller, Pod

Step 1 Before diving into Kubernetes deployment, let's delete our service, controller, pods. To delete the service execute the following command:

```
$ kubectl delete service echoserver
service "echoserver" deleted
```

Step 2 To delete the replication controller and its pods:

```
$ kubectl delete replicationcontroller echoserver
replicationcontroller "echoserver" deleted
```

Step 3 Check that there are no running pods:

```
$ kubectl get pods
```

Note that if you want to delete just a replication controller without deleting any of its pods, use the option `--cascade=false`.

4.7. Create a Deployment

Step 1 The simplest way to create a new deployment for a single-container pod is to use `kubectl run`:

```
$ kubectl run echoserver \
--image=gcr.io/google_containers/echoserver:1.4 \
--port=8080 \
```

```
--replicas=2
deployment "echoserver" created
```

Step 2 Check pods:

```
$ kubectl get pods
NAME                          READY     STATUS    RESTARTS   AGE
echoserver-722388366-5z1bh    1/1       Running   0          1m
echoserver-722388366-nhfb1    1/1       Running   0          1m
```

Step 3 To access the echoserver from the lab, create a new service using `kubectl expose deployment`:

```
$ kubectl expose deployment echoserver --type=NodePort
service "echoserver" exposed
```

To get the exposed port number execute:

```
$ kubectl describe services/echoserver | grep ^NodePort
NodePort:           <unset> 30512/TCP
```

Remember the port number for the next step.

Step 4 Check that the echoserver is accessible:

```
$ curl http://localhost:<port>
CLIENT VALUES:
...
```

Step 5 Let's change the number of replicas in the deployment. Use `kubectl edit` to open an editor and change the number of replicas to 3:

```
$ kubectl edit deployment echoserver
# edit the deployment definition, change replicas to 3
deployment "echoserver" edited
```

Step 6 View the deployment details:

```
$ kubectl describe deployment echoserver
Name:                   echoserver
Namespace:              default
Labels:                 run=echoserver
Selector:               run=echoserver
Replicas:               3 updated | 3 total | 3 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
OldReplicaSets:                 <none>
NewReplicaSet:                  echoserver-722388366 (3/3 replicas created)
...
```

Step 7 Check that there are 3 running pods:

```
$ kubectl get pods
NAME                          READY     STATUS    RESTARTS   AGE
```

```
echoserver-722388366-5p5ja   1/1       Running   0          1m
echoserver-722388366-5z1bh   1/1       Running   0          1h
echoserver-722388366-nhfb1   1/1       Running   0          1h
```

Step 8 Use `kubectl rollout history deployment` to see revisions of the deployment:

```
$ kubectl rollout history deployment echoserver
deployments "echoserver":
REVISION    CHANGE-CAUSE
1           <none>
```

Step 9 Now we want to replace our echoserver with a new implementation. We want to use a new image based on `busybox`. Edit the deployment:

```
$ kubectl edit deployment echoserver
```

Step 10 Change the `image` value to:

```
image: gcr.io/google-containers/busybox
```

Step 11 And add a new `command` field just after the `image`:

```
image: gcr.io/google-containers/busybox
command: ['nc', '-p', '8080', '-l', '-l', '-e', 'echo', 'hello world!']
```

Step 12 Check the deployment status:

```
$ kubectl describe deployment echoserver
...
Replicas:           3 updated | 3 total | 3 available | 0 unavailable
...
```

Step 13 Check that the echoserver works (use the port number from the step 3):

```
$ curl http://localhost:<port>
hello world!
```

Step 14 The deployment controller replaced all of the pods by new ones, one by one. Let's check the revisions:

```
$ kubectl rollout history deployment echoserver
deployments "echoserver":
REVISION    CHANGE-CAUSE
1           <none>
2           <none>
```

Step 15 After that, we decided that the new implementation does not work as expected (we wanted echoserver, not a hello world application). Let's undo the last change:

```
$ kubectl rollout undo deployment echoserver
deployment "echoserver" rolled back
```

Step 16 Check the deployment status:

```
$ kubectl describe deployment echoserver
...
Replicas:           3 updated | 3 total | 3 available | 0 unavailable
...
```

Step 17 Check that the echoserver works (use the port number from the step 3):

```
$ curl http://localhost:<port>
CLIENT VALUES:
...
```

Step 18 Delete the deployment:

```
$ kubectl delete deployment echoserver
deployment "echoserver" deleted
```

We have successfully rolled back the deployment and our pods are based on the `echoserver` image again.

4.8. Create a Job

Step 1 Define a new job in the file `myjob.yaml`:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: myjob
spec:
  completions: 5
  parallelism: 1
  template:
    metadata:
      name: myjob
    spec:
      containers:
      - name: myjob
        image: busybox
        command: ["sleep", "20"]
      restartPolicy: Never
```

This job is based on the image `busybox`. It waits 20 seconds, then exists. We requested 5 successful completions with no parallelism.

Step 2 Create a new job:

```
$ kubectl create -f myjob.yaml
job "myjob" created
```

Step 3 Let's watch the job being executed and the results of each execution:

```
$ kubectl get jobs --watch
NAME      DESIRED   SUCCESSFUL   AGE
myjob     5         1            1m
...
```

Step 4 If we're interested in more details about the job:

```
$ kubectl describe jobs myjob
...
Pods Statuses:      1 Running / 0 Succeeded / 0 Failed
...
```

Step 5 And finally:

```
$ kubectl describe jobs myjob
...
Pods Statuses:      0 Running / 5 Succeeded / 0 Failed
...
```

Step 6 After that, the job can be deleted:

```
$ kubectl delete job myjob
job "myjob" deleted
```

Checkpoint

- Install a single-node Kubernetes cluster in a virtual machine
- Create a pod, replication controller, service, deployment, job

5. Cohesive Application Deployments

In this lab, we will declare and create a cohesive multi-application stack with services through Kubernetes.

Chapter Details	
Chapter Goal	Declare, deploy, and verify a cohesive application deployment
Chapter Sections	*5.1. Application Deployment Architecture* *5.2. Kubernetes Secrets* *5.3. Create a Persistent Volume* *5.4. Create the MySQL Deployment* *5.5. Create a Service for MySQL* *5.6. Create the Wordpress Deployment* *5.7. Create a Service for Wordpress*

5.1. Application Deployment Architecture

In this lab we will deploy the Wordpress content management system with a MySQL backend. Both applications will reside within separate Pods. We will declare and create a Service for MySQL which Wordpress will use to connect to the database. We will also use a Kubernetes secret to hold our database connection information such as username, passwords, and database name.

Since MySQL is persisting the data on behalf of Wordpress, we will provision a persistent volume that will be used to house the data for MySQL.

To be able to access the Wordpress CMS from outside the Kubernetes cluster, we will also create a Service for Wordpress to allow for easy access. Alright, let's get to it.

5.2. Kubernetes Secrets

Kubernetes secrets allow users to define sensitive information outside of containers and expose that information to containers through environment variables as well as files within Pods. In this section we will declare and create secrets to hold our database connection information that will be used by Wordpress to connect to its backend database.

Step 1 Open up two terminal windows. We will use one window to generate encoded strings that will contain our sensitive data. The other window will be used to create the secrets YAML declaration.

Step 2 In the first terminal window, execute the following commands to encode our strings:

```
$ echo -n "wordpress_db" | base64
d29yZHByZXNzX2Ri

$ echo -n "wordpress_user" | base64
d29yZHByZXNzX3VzZXI=

$ echo -n "wordpress_pass" | base64
d29yZHByZXNzX3Bhc3M=

$ echo -n "rootpass123" | base64
cm9vdHBhc3MxMjM=
```

Step 3 In the second terminal window, create a directory where we will store our declarations and create a file named `wordpress-db-secrets.yaml` with the contents below, making sure to copy and paste the encoded values from the previous step into the appropriate fields:

```
$ mkdir -p ~/k8s-artifacts/wordpress
$ cd ~/k8s-artifacts/wordpress
$ vim wordpress-db-secrets.yaml

apiVersion: v1
kind: Secret
metadata:
  name: wordpress-db-secrets
type: Opaque
data:
  dbname: d29yZHByZXNzX2Ri
  dbuser: d29yZHByZXNzX3VzZXI=
```

```
  dbpassword: d29yZHByZXNzX3Bhc3M=
  mysqlrootpassword: cm9vdHBhc3MxMjM=
```

Step 4 Create the secret:

```
$ kubectl create -f wordpress-db-secrets.yaml
secret "wordpress-db-secrets" created
```

Step 5 Verify creation and get details:

```
$ kubectl get secrets
NAME                  TYPE                                  DATA      AGE
default-token-ahhjz   kubernetes.io/service-account-token   3         37d
wordpress-db-secrets  Opaque                                4         3h

$ kubectl describe secrets/wordpress-db-secrets
Name:           wordpress-db-secrets
Namespace:      default
Labels:         <none>
Annotations:    <none>

Type:   Opaque

Data
====
dbname:             12 bytes
dbpassword:         14 bytes
dbuser:             14 bytes
mysqlrootpassword:  11 bytes
```

5.3. Create a Persistent Volume

We will use a persistent volume to provide the underlying storage target for our MySQL database. In our environment, we will use a HostPath device, which will just pass a directory from the host into a container for consumption.

Notes

If you are using a hosted Kubernetes environment, such as on Google Cloud Platform, persistent volumes can be created as GCE Persistent disks rather than HostPath. One downfall to HostPath is that it only works with single node Kubernetes clusters, as there is no guarantee Pods will get created where storage exists when multiple node deployments are used.

Step 1 Create a directory we will use as our HostPath persistent disk:

```
$ sudo mkdir -p /data/mysql-wordpress
```

Step 2 Create a file under `~/k8s-artifacts/wordpress` named `mysql-persistentvol.yaml` with the contents below:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv
  labels:
    vol: mysql
spec:
  capacity:
    storage: 5Gi
  accessModes:
  - ReadWriteOnce
  hostPath:
    path: /data/mysql-wordpress
```

Step 3 Create a Kubernetes persistent volume and verify its status is set to `Available`:

```
$ kubectl create -f mysql-persistentvol.yaml
persistentvolume "mysql-pv" created

$ kubectl describe pv/mysql-pv
Name:           mysql-pv
Labels:         vol=mysql
Status:         Available
Claim:
Reclaim Policy: Retain
Access Modes:   RWO
Capacity:       5Gi
Message:
Source:
  Type:         HostPath (bare host directory volume)
  Path:         /data/mysql-wordpress
No events.
```

5.4. Create the MySQL Deployment

We are now ready to declare and have Kubernetes create our deployment for MySQL.

Step 1 Create a file named `mysql-deployment.yaml` in the `~/k8s-artifacts/wordpress` directory with the contents below:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: mysql-pv-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  selector:
    matchLabels:
```

```
      vol: "mysql"
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: mysql-deployment
spec:
  replicas: 1
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: mysql
        track: production
    spec:
      containers:
      - name: "mysql"
        image: "mysql:5.6"
        ports:
        - containerPort: 3306
        volumeMounts:
        - mountPath: "/var/lib/mysql"
          name: mysql-pd
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: wordpress-db-secrets
              key: mysqlrootpassword
        - name: MYSQL_USER
          valueFrom:
            secretKeyRef:
              name: wordpress-db-secrets
              key: dbuser
        - name: MYSQL_PASSWORD
          valueFrom:
            secretKeyRef:
              name: wordpress-db-secrets
              key: dbpassword
        - name: MYSQL_DATABASE
          valueFrom:
            secretKeyRef:
              name: wordpress-db-secrets
              key: dbname
      volumes:
        - name: mysql-pd
          persistentVolumeClaim:
            claimName: mysql-pv-claim
```

Alright, so there is quite a bit of stuff happening in the above YAML. Let us try to break it down piece by piece.

We are declaring two resources within this single YAML document. The first resource is a persistent volume claim to our previously created persistent volume. We are making the association to the persistent volume through our label selector.

The second resource we are declaring is our deployment for MySQL. We are specifying our standard information such as container name, image to use, and exposed port to access. We are then mounting a volume to the `/var/lib/mysql` directory in the container, where the volume to mount is named `mysql-pd` and is declared at the bottom of this document.

We are also declaring environment variables to initialize. The MySQL image we are using that is available on Docker Hub supports environment variable injection. The four environment variables we are initializing are defined and used within the Docker image itself. The values we are setting these environment variables to are all referencing different keys we set in our Secret earlier on. When this container starts up, we will automatically have MySQL configured with the desired root user password, and we will also have the database for Wordpress created with appropriate access granted for our Wordpress user.

Step 2 Create the resources and verify everything was created successfully:

```
$ kubectl create -f mysql-deployment.yaml
persistentvolumeclaim "mysql-pv-claim" created
deployment "mysql-deployment" created

$ kubectl get pv
NAME      CAPACITY  ACCESSMODES   STATUS  CLAIM                   REASON    AGE
mysql-pv  5Gi       RWO           Bound   default/mysql-pv-claim            14m

$ kubectl get pvc
NAME            STATUS  VOLUME    CAPACITY  ACCESSMODES AGE
mysql-pv-claim  Bound   mysql-pv  0                     2m

$ kubectl get deployments
NAME              DESIRED   CURRENT   UP-TO-DATE  AVAILABLE   AGE
mysql-deployment  1         1         1           1           2m
```

Step 3 Let's verify the device was properly mapped into the container and the database was successfully created:

```
$ kubectl get pods
NAME                    READY   STATUS    RESTARTS  AGE
mysql-deployment-<id>   1/1     RUNNING   0         2m

$ kubectl exec -it mysql-deployment-<id> bash

root@mysql-deployment:/# mount | grep /var/lib/mysql
tmpfs on /var/lib/mysql type tmpfs (rw,relatime,size=917852k)

root@mysql-deployment:/# mysql -u root -p
Enter password: rootpass123

mysql> show databases;
+-----------------+
```

```
| Database        |
+-----------------+
...
| wordpress_db    |
+-----------------+

mysql> show grants for wordpress_user;
+---------------------------------------------------------------------------+
| Grants for wordpress_user@%                                               |
+---------------------------------------------------------------------------|
| GRANT USAGE ON *.* TO 'wordpress_user'@'%' IDENTIFIED BY PASSWORD <pass>  |
| GRANT ALL PRIVILEGES ON `wordpress_db`.* TO 'wordpress_user'@'%'          |
+---------------------------------------------------------------------------+
```

5.5. Create a Service for MySQL

As we know, Pods are ephemeral. They come and go and needed, with each newly created Pod receiving a new and different IP address. Because of this, for our Pods to reliably communicate with one another, we cannot staticly configure them to talk directly to a Pod over its assigned IP address. We need an IP address that is decoupled from that of a Pod and that never changes, and this is exactly what Kubernetes services offer.

In this section we will go ahead and declare and create a service for MySQL. Let's get to it!

Step 1 Create a file named `mysql-service.yaml` in the `~stack/k8s-artifacts/wordpress` directory and populate it with the contents below:

```
apiVersion: v1
kind: Service
metadata:
  name: mysql-internal
spec:
  ports:
    - port: 3306
      protocol: TCP
      targetPort: 3306
  selector:
    app: mysql
```

Step 2 Create the service and verify its properties:

```
$ kubectl create -f mysql-service.yaml
service "mysql-internal" created

$ kubectl describe svc/mysql-internal
Name:               mysql-internal
Namespace:          default
Labels:             <none>
Selector:           app=mysql
Type:               ClusterIP
IP:                 ...
Port:               <unset> 3306/TCP
Endpoints:          172.17.0.3:3306
Session Affinity:   None
```

```
No events.

$ kubectl get pods -o wide
NAME                     READY   STATUS    RESTARTS  AGE   IP
mysql-deployment-<id>    1/1     RUNNING   0         30m   172.17.0.3
```

From the output above, we can verify the service was correctly mapped to to the pod for our MySQL deployment in that the `Endpoints` IP for the service aligns with the IP for the MySQL Pod.

5.6. Create the Wordpress Deployment

Now that MySQL is deployed, persisting data to a volume, and is accessible over a Kubernetes service, we are ready to deploy our application - Wordpress. Wordpress is a very popular content management system written in PHP that has several different image versions available on Docker hub. We will use one of the pre-existing images that bundles Wordpress together with Apache, PHP, and PHP modules in our exercise.

Step 1 Let's first declare our Wordpress deployment. In the `~/k8s-artifacts/wordpress` directory populate a file named `wordpress-deployment.yaml` with the following contents:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: wordpress-deployment
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: wordpress
        track: production
    spec:
      containers:
      - name: "wordpress"
        image: "wordpress:4.5-apache"
        ports:
        - containerPort: 80
        env:
        - name: WORDPRESS_DB_HOST
          value: "mysql-internal"
        - name: WORDPRESS_DB_USER
          valueFrom:
            secretKeyRef:
              name: wordpress-db-secrets
              key: dbuser
        - name: WORDPRESS_DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: wordpress-db-secrets
```

```
            key: dbpassword
      - name: WORDPRESS_DB_NAME
        valueFrom:
          secretKeyRef:
            name: wordpress-db-secrets
            key: dbname
```

One of the key things we are doing in the YAML above is initializing the environment variable `WORDPRESS_DB_HOST` to a value of `mysql-internal`. This is how we are telling the Wordpress application to access its database through the Kubernetes service we created in the previous section.

When we created that service, Kubernetes created a DNS record mapping the IP of the service to a name equal to the name of the service itself, which as you recall was `mysql-internal`.

Step 2 Create and verify the deployment:

```
$ kubectl create -f wordpress-deployment.yaml
deployment "wordpress-deployment" created

$ kubectl get deployments
NAME                  DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
mysql-deployment      1         1         1            1           45m
wordpress-deployment  3         3         3            3           5m
```

Step 3 Get a list of created Pods:

```
$ kubectl get pods
NAME                         READY   STATUS    RESTARTS   AGE
mysql-deployment-425         1/1     Running   0          45m
wordpress-deployment-hq4     1/1     Running   0          6m
wordpress-deployment-jnr     1/1     Running   0          6m
wordpress-deployment-l13     1/1     Running   0          6m
```

Make note of the name of one of the Wordpress Pods from the output above.

Step 4 Execute a shell within one of the Wordpress Pods found in the previous step:

```
$ kubectl exec -it <pod name> bash

root@wordpress# getent hosts mysql-internal
10.0.0.248      mysql-internal.default.svc.cluster.local
# Your IP may be different
```

The above output verifies that mysql-internal can be resolved through DNS to the ClusterIP address that was assigned to the MySQL service. Good stuff!

Now let's verify Wordpress was properly configured:

```
root@wordpress# grep -i db /var/www/html/wp-config.php
define('DB_NAME', 'wordpress_db');
define('DB_USER', 'wordpress_user');
define('DB_PASSWORD', 'wordpress_pass');
define('DB_HOST', 'mysql-internal');
...
```

That all looks good as well. Nice work champ!

5.7. Create a Service for Wordpress

The final thing required is to expose the Wordpress application to external users. For this, we again will need a service. In this example, we will expose a high numbered port on the Node running our application, and DNAT it to port 80 of our container. It will allow us to access the application, but it probably is not the approach one would take in production, especially if Kubernetes is hosted by a service provider.

Kubernetes can integrate with Load Balancing services offered by platforms such as GCE and AWS. If you are using either of those, then that would be one approach to take to take advantage of the load balancing functionality offered in those platforms.

Step 1 Create a file named `wordpress-service.yaml` in the `~/k8s-artifacts/wordpress` directory. Populate the file with the contents below:

```
apiVersion: v1
kind: Service
metadata:
  name: wordpress-service
  labels:
    app: wordpress
    track: production
spec:
  type: NodePort
  ports:
    - port: 80
      nodePort: 30080
  selector:
    app: wordpress
    track: production
```

Step 2 Create the service and verify its status:

```
$ kubectl create -f wordpress-service.yaml
service "wordpress-service" created

$ kubectl describe svc/wordpress-service
Name:               wordpress-service
Namespace:          default
Labels:             app=wordpress
                    track=production
Selector:           app=wordpress,track=production
Type:               NodePort
IP:                 ...
Port:               <unset> 80/TCP
NodePort:           <unset> 30080/TCP
Endpoints:          ...:80,...:80,...:80
Session Affinity:   None
No events.
```

Step 4 Open up your browser and navigate to `http://<lab IP>:30080`. You can follow the installation wizard to get Wordpress up and running through the browser.

Congratulations on deploying a cohesive multi-pod application stack with Kubernetes!

Checkpoint

- Create and use a secret
- Declare containers with environment variables
- Declare a persistent volume, persistent volume claim, and volume mount
- Link pods through services

http://training.mirantis.com/