Introduction

Similar to virtualization, containers provide a method of encapsulating applications so that several containers can be run on a single physical machine without interfering with each other. However, the method and level of the isolation differs from virtualization. The physical host machine runs an operating system that manages the hardware directly. For virtualization, each virtual machine runs a complete operating system. In a container, the operating system is provided by the physical host machine. The application in the container interfaces with the host kernel through the container engine, which provides isolation between the containers. Each container can have its own view of the network, file system, processes, users, and groups but uses the same OS kernel as the host.

Docker is a popular system for implementing containers. It was originally developed using Linux containers (LXC) to implement the containerization. This has been replaced with a custom library written in Go. Both systems use the Linux kernel support for cgroups and namespaces. Cgroups allow the resource usage of a process to be limited. Namespaces allow the management of the resources visible to a process. The Linux kernel supports namespaces for the file system, process IDs, network devices, inter-process communication (IPC), host names and domain names, user and group IDs, cgroups, and time. Each process is attached to a namespace for each type of resources, and that namespace defines what is visible to the process.

Originally, because of its Linux roots, Docker supported running only Linux-based applications on Linux host machines. This requirement has been relaxed in recent releases. It is now also possible to run Windows containers on Windows hosts. There is even support for running Linux containers on Windows hosts, but the implementation here actually uses a simplified virtual machine running a Linux kernel.

The Docker architecture is a client-server architecture. The host machine runs a server process, the Docker daemon. This process manages all the docker objects on the host and listens for requests via an API. There are several kinds of managed objects. Here are some of the most important:

- An **image** is a read-only set of instructions for how to construct a container. An image can be based on an already existing image.
- A **container** is an image instance that can be run.
- A **volume** is a read-write object to hold mutable data for a container.
- A **network** is a communication channel between containers. There are different kinds of networks, some of which allow communication with the host physical network.

The Docker daemon obtains images from a registry. The user can interact with the Docker daemon using the command line Docker client. The Docker client does not need to be running on the same machine as the Docker daemon.

In this lab scenario, some of your company's web applications are seeing increased use and you are investigating ways of scaling the application. One possible solution is using Docker containers. The application can be scaled horizontally by having multiple instances of the app running. By using containers, the number of instances can be easily scaled up and down depending on the demand. Your team is tasked with the initial investigation into using Docker.

Lab Overview

This lab has **three** parts, which should be completed in the order specified.

1. In the first part of the lab, you will create a Docker image and a Docker container and then run the container.

2. In the second part of the lab, you will create and run an application that uses two images that work together.

3. In the third part of the lab, you will define and run your multi-container application using the Docker Compose tool.

Finally, you will explore the virtual environment on your own. You will answer questions and complete challenges that allow you to use the skills you learned in the lab to conduct independent, unguided work – similar to what you will encounter in a real-world situation.

Learning Objectives

Upon completing this lab, you will be able to:

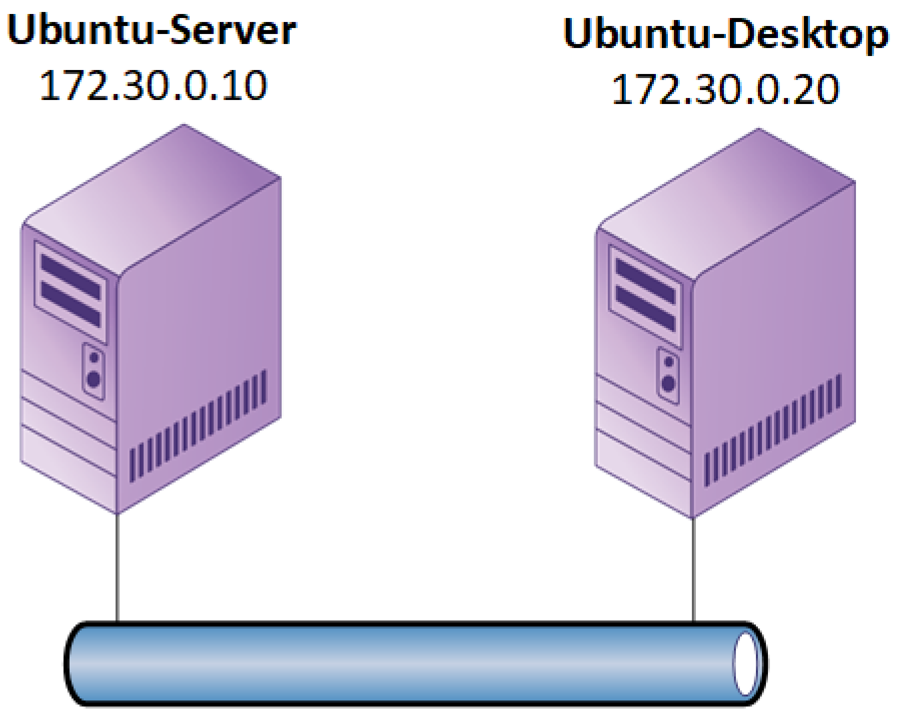
1. Explain the difference between containerization and virtualization.

2. Build and run Docker containers using the command line.
3. Configure a Docker image using a Dockerfile.
4. Use the Docker command line to manage networks.
5. Use Docker Compose to manage a multi-container application.

Topology

This lab contains the following virtual machines. Please refer to the network topology diagram below.

- Ubuntu-Desktop
- Ubuntu-Server



Tools and Software

The following software and/or utilities are required to complete this lab. Students are encouraged to explore the Internet to learn more about the products and tools used in this lab.

- Docker
- vim
- xargs

Deliverables

Upon completion of this lab, you are required to provide the following deliverables to your instructor:

Hands-On Demonstration

1. Lab Report file, including screen captures of the following:

- the successful build of the test image
- the working directory, file listing, and environment variable value in the testc container
- the running db container
- the command creating the network and the resulting id
- the Django page
- the output from the compose up command
- your docker run command and the contents of the file_1.txt

2. Any additional information as directed by the lab:

- None

Challenge and Analysis

1. Lab Report file, including screen captures of the following:

- the successful build of the web2 image
- the directory listing served on port 8080
- the creation of the web, web2, and db containers in the docker compose output
- the directory listing served on port 8080

2. Any additional information as directed by the lab:

- None

Hands-On Demonstration

Note: In this section of the lab, you will follow a step-by-step walk through of the objectives for this lab to produce the expected deliverables.

1. **Review** the **Tutorial**.

   Frequently performed tasks, such as making screen captures and downloading your Lab Report, are explained in the Cloud Lab Tutorial. The Cloud Lab Tutorial is available from the User menu in the upper-right corner of the Student Dashboard. You should review these tasks before starting the lab.

2. **Proceed** with **Part 1**.

Part 1: Creating and Running a Docker Container

Note: A Docker container is based on an image. The image, in turn, is defined by a Dockerfile. A Dockerfile provides a way to define the sequence of steps needed to construct an image. To get a running container, a sequence of steps must be carried out:

- The image is built using the steps defined in the Dockerfile.
- The container is created using an image and additional configuration information.
- The container can then be started with a specified command to run.

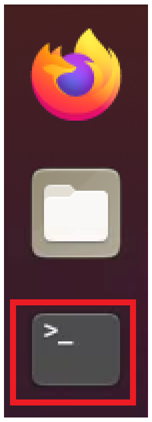
Each of these steps can be accomplished using the appropriate command for the Docker client. For many typical situations, the last two steps can also be combined using the single "docker run" command.

For your use case, it is also going to be important to have control over the registry that the images are pulled from. Another part of your team has set up the Docker-Server machine so that it is running a Docker registry. This registry is available at port 5000 at 172.30.0.10. The first step of your investigation is to understand how a Dockerfile can be used to define the image for a container.

1. If necessary, on the Ubuntu-Desktop login screen, **type the following credentials** and **press Enter** to log in.

   Username: `user`
   Password: `password`

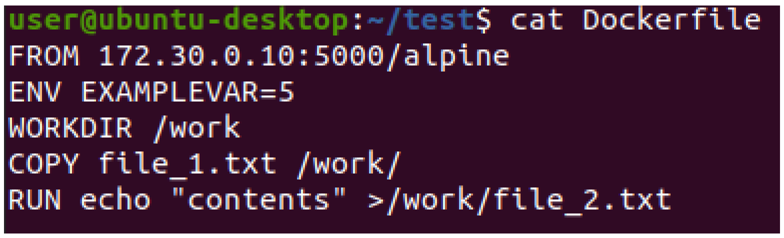
2. On the sidebar, **click** the **Terminal icon** to open a new terminal.



Terminal Icon

3. In the terminal, **type** `cd test` and **press Enter** to change the current directory.

4. In the terminal, **type** `cat Dockerfile` and **press Enter** to see the contents of the Dockerfile.



```
user@ubuntu-desktop:~/test$ cat Dockerfile
FROM 172.30.0.10:5000/alpine
ENV EXAMPLEVAR=5
WORKDIR /work
COPY file_1.txt /work/
RUN echo "contents" >/work/file_2.txt
```

Dockerfile contents

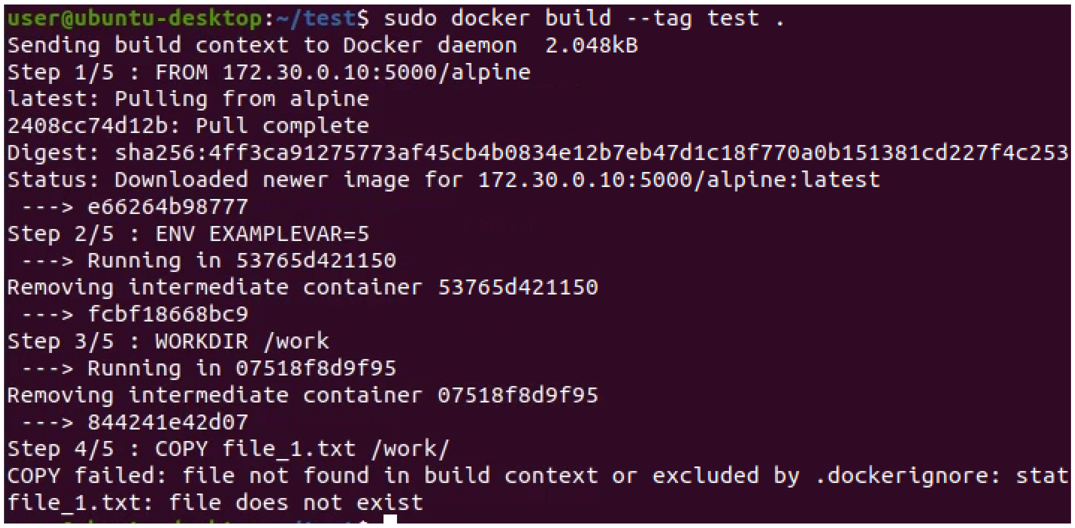
Note: The first line begins with FROM. This indicates the initial image to use as the base for a new image to be built. In this case, the starting point is the latest version of the alpine image from the repository at 172.30.0.10 port 5000. The general form for an image is [optional registry]/<image name>:<image version>. If the registry is not specified, it will default to the Docker Hub registry. If the image version is not given, it will default to "latest."

The second line starts with ENV. This line defines an environment variable that should be set in the container. The WORKDIR line sets the working directory for the lines after it. In this case, the working directory becomes /work in the container.

The COPY line indicates that a file should be copied from the build environment to the container. The first path is relative to the working directory in the build environment. The second path is relative to the current WORKDIR from the Dockerfile. For this example, the file /home/user/test/file_1.txt will be copied to /work/file_1.txt in the container.

The final line gives a command that should be run in the context of the container to change its state. Thus, the line in the example will create a file /work/file_2.txt in the container that has the text "contents." To actually build the image, you use the "build" command. In this lab, all the Docker commands will be prefixed with *sudo*. By default, the communication with the Docker daemon happens over a Unix socket that is owned by root. That means interacting with the daemon requires root privileges.

5. In the terminal, **type** **`sudo docker build --tag test .`** and **press Enter** to attempt to build the image using the Dockerfile. **Type** **`password`** and **press Enter** if asked for the sudo password.



```
user@ubuntu-desktop:~/test$ sudo docker build --tag test .
Sending build context to Docker daemon  2.048kB
Step 1/5 : FROM 172.30.0.10:5000/alpine
latest: Pulling from alpine
2408cc74d12b: Pull complete
Digest: sha256:4ff3ca91275773af45cb4b0834e12b7eb47d1c18f770a0b151381cd227f4c253
Status: Downloaded newer image for 172.30.0.10:5000/alpine:latest
 ---> e66264b98777
Step 2/5 : ENV EXAMPLEVAR=5
 ---> Running in 53765d421150
Removing intermediate container 53765d421150
 ---> fcbf18668bc9
Step 3/5 : WORKDIR /work
 ---> Running in 07518f8d9f95
Removing intermediate container 07518f8d9f95
 ---> 844241e42d07
Step 4/5 : COPY file_1.txt /work/
COPY failed: file not found in build context or excluded by .dockerignore: stat
file_1.txt: file does not exist
```

Command to build test image

Note: The Docker command has a few parts.

- sudo
  required to communicate with the daemon
- docker build
  specifies the command to perform
- --tag test
  indicates a label to use for naming the resulting image
- .
  indicates the directory containing the Dockerfile (the current working directory)

The Docker build command proceeds to follow the instructions in the Dockerfile. In the output, you should see lines labeled Step X/5. For this simple Dockerfile, each of the four uncommented lines corresponds directly to a step. The output from the command indicates the actions that were taken.

For step 1, the latest alpine image was retrieved from the local registry. For step 2, the environment variable was added. Notice that this step has a line "Running in ...." and then "Removing intermediate container ...." The hexadecimal string is a container ID. When building the image, the Docker daemon creates a container and takes the action specified in the Dockerfile, and if the container is no longer needed, it is then removed. For step 3, the working directory is changed.

Step 4 failed in this attempt. The message indicates that file_1.txt was not found. You can check and see that there is in fact no file_1.txt in the current working directory.

6. In the terminal, **type** `ls` and **press Enter** to list the files.

Note: You should see only one file, Dockerfile. Therefore, you need to create a file with the appropriate name to enable the build to succeed.

7. In the terminal, **type** `echo "example file content" >file_1.txt` and **press Enter** to create the file.

8. In the terminal, **type** `sudo docker build --tag test .` and **press Enter** to build the image.

9. **Make a screen capture** showing the **successful build of the test image**.

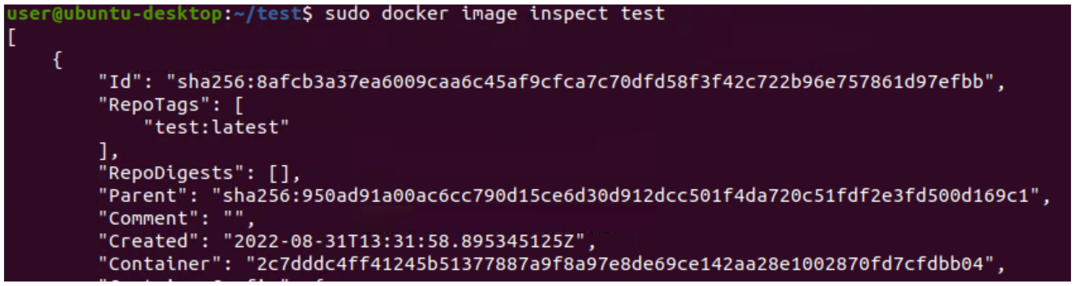
Note: This time the build should work. Notice that the first three steps look different. The alpine image has already been retrieved, so it does not need to be retrieved again. Then the next two steps indicate that a cached image is being used. Since the starting point for each line is the same and nothing has changed in the contents of those lines, the images that result from those steps will be the same. The Docker daemon caches intermediate images from build steps when possible.

This time, step 4 found file_1.txt, so it was able to copy it into the image. Then the final step also succeeds. The final two lines of the output give the image ID of the result and then indicate that the requested tag "test" was applied with the default implied version "latest."

10. In the terminal, **type** `sudo docker image ls` and **press Enter** to list the images that are available.

Note: You should see two images: alpine (which was retrieved from the registry) and test (which was just created). It is also possible to request detailed information about an image.

11. In the terminal, **type** `sudo docker image inspect test` and **press Enter** to show detailed information about the test.



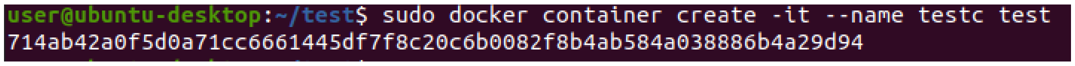
```
user@ubuntu-desktop:~/test$ sudo docker image inspect test
[
    {
        "Id": "sha256:8afcb3a37ea6009caa6c45af9cfca7c70dfd58f3f42c722b96e757861d97efbb",
        "RepoTags": [
            "test:latest"
        ],
        "RepoDigests": [],
        "Parent": "sha256:950ad91a00ac6cc790d15ce6d30d912dcc501f4da720c51fdf2e3fd500d169c1",
        "Comment": "",
        "Created": "2022-08-31T13:31:58.895345125Z",
        "Container": "2c7dddc4ff41245b51377887a9f8a97e8de69ce142aa28e1002870fd7cfdbb04",
```

Command to inspect the test image

Note: You should see quite a few lines of JSON. These are all the settings related to the test image. They can be adjusted by command line options used with the Docker build command. You can read about the many options available here.

Now that you have an image, you can create a container. This is done using the "container create" command.

12. In the terminal, **type** `sudo docker container create -it --name testc test` and **press Enter** to create a container from the image.



```
user@ubuntu-desktop:~/test$ sudo docker container create -it --name testc test
714ab42a0f5d0a71cc6661445df7f8c20c6b0082f8b4ab584a038886b4a29d94
```

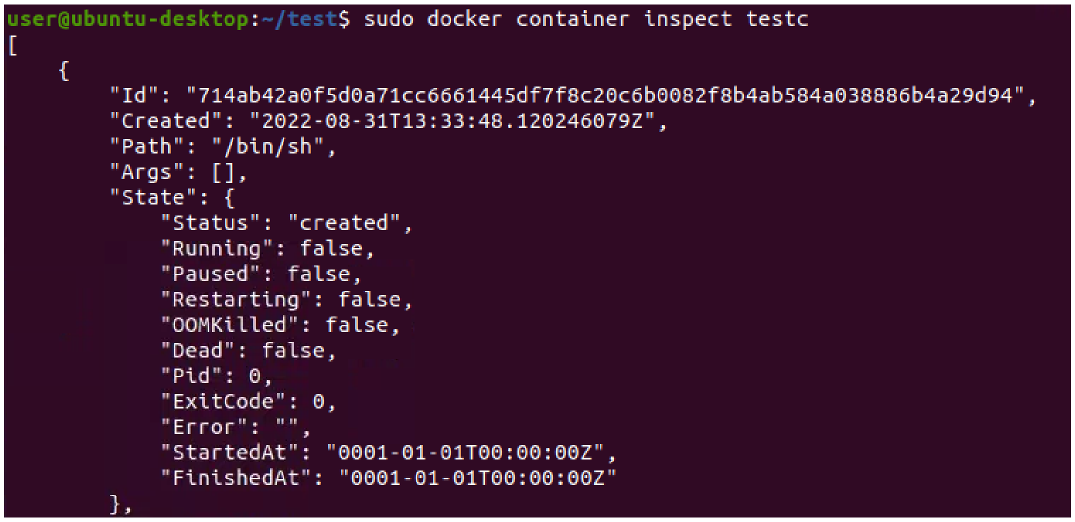
Command to create the test container

Note: The "-it" flags (which could also be typed "-i -t") indicate that when the container is run it should be interactive (-i) and should allocate a pseudo TTY (-t). This will allow you to interact directly with the container using a shell. The "--name testc" arguments give a human-friendly name to the container. The final argument "test" indicates the image to use.

13. In the terminal, **type** **`sudo docker container ls --all`** and **press Enter** to list the containers.

Note: The flag --all is needed because the command shows only running containers by default. In the results you see a COMMAND column that shows "/bin/sh." The Dockerfile did not specify the command to run in the container, so the default shell is used as the command. There is also an inspect command for containers.

14. In the terminal, **type** **`sudo docker container inspect testc`** and **press Enter** to view details about the container.

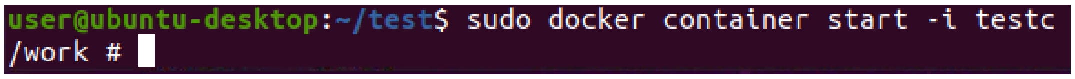


```
user@ubuntu-desktop:~/test$ sudo docker container inspect testc
[
    {
        "Id": "714ab42a0f5d0a71cc6661445df7f8c20c6b0082f8b4ab584a038886b4a29d94",
        "Created": "2022-08-31T13:33:48.120246079Z",
        "Path": "/bin/sh",
        "Args": [],
        "State": {
            "Status": "created",
            "Running": false,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 0,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "0001-01-01T00:00:00Z",
            "FinishedAt": "0001-01-01T00:00:00Z"
        },
```

Command to inspect the test container

Note: Again, the output is many lines of JSON. As before, many of these settings can be controlled by the options given to the "container create" command. You can read about them here. Finally, you can start the container using the "container start" command.

15. In the terminal, **type** **`sudo docker container start -i testc`** and **press Enter** to start the container.



```
user@ubuntu-desktop:~/test$ sudo docker container start -i testc
/work #
```

Command to start the test container

Note: The -i argument is needed again here to indicate that you want to interact with the container. You should see a shell prompt "/work #." This is a shell running inside the container. You can verify that the environment matches what was defined in the Dockerfile using simple shell commands.

16. In the terminal, **type** `pwd` and **press Enter** to see the working directory.

17. In the terminal, **type** `ls` and **press Enter** to list the files.

18. In the terminal, **type** `echo $EXAMPLEVAR` and **press Enter** to check for the environment variable.

Note: You should see that the present working directory is /work, it contains two files ("file_1.txt" and "file_2.txt"), and the environment variable has the value 5.

19. **Make a screen capture** showing the **working directory, file listing, and environment variable value**.

20. In the terminal, **type** `exit` and **press Enter** to exit the container shell.

Note: The container is no longer running, but it still exists.

21. In the terminal, **type** `sudo docker container ls --all` and **press Enter** to list all

containers.

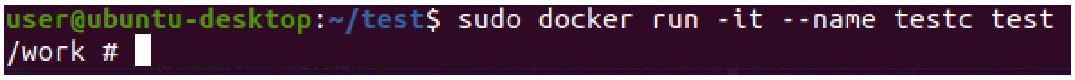
Note: You should see the container named "testc"; its status shows that it is not running. The "container rm" command can be used to remove containers.

22. In the terminal, **type** `sudo docker container rm testc` and **press Enter** to remove the testc container.

23. In the terminal, **type** `sudo docker container ls --all` and **press Enter** to verify that the container no longer exists.

Note: There are no containers anymore. As noted, the multi-step process of creating a container and then starting the container can often be carried out using a single "run" command.

24. In the terminal, **type** `sudo docker run -it --name testc test` and **press Enter** to run the container.



```
user@ubuntu-desktop:~/test$ sudo docker run -it --name testc test
/work #
```

Command to create and run a container

25. In the terminal, **type** `cat file_1.txt` to output the content of file_1.txt to the terminal.

26. **Make a screen capture** showing **both your docker run command and the contents of the file_1.txt**.

27. **Exit** the container.

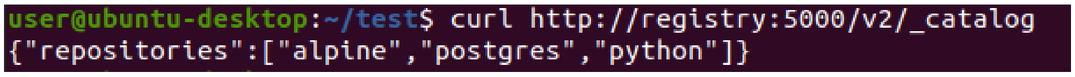
Part 2: Configuring an Application with Multiple Containers

Note: A Docker container runs a single command. This is an important distinction from virtual machines. A virtual machine runs the full operating system and has many processes running. Many containers contain only a single running process. However, this is not an actual limit. The initial command can be designed to start multiple processes. Docker will cause the container to terminate when the first process that was started terminates. In general, Docker containers should be designed to run a single "process." In the case of something like a web server, that might mean several processes are actually running within the container.
Many web applications and services are built out of multiple parts. Two very common components are a database and a web server. It is also common for the application itself to be divided into multiple cooperating services. When designing an application for containerization, the recommended practice is to have each of the cooperating services running in separate containers and communicating with each other rather than trying to get them all running inside a single container. This helps to preserve the isolation among the pieces so that failures in one part do not automatically cause failures in another.
Docker containers communicate using networking. There are several kinds of network that are available in the base Docker architecture.

- The default **bridge** network driver enables containers on the same host to communicate.
- The **host** network driver makes the container use the host networking directly.
- The **overlay** driver allows containers on different hosts to communicate without requiring changes to the host-level routing.
- The **ipvlan** and **macvlan** drivers give you control over the layer-three (ipvlan) or layer-two (macvlan) addresses of the container network devices.
- You can also specify **none** as the networking driver to disable networking for a container.

Your company frequently uses Django-based applications with Postgres as the database backend. First, you will manually create Docker containers with these two services so that they can communicate. For your testing, you are running all the containers on the same host, so bridge networking is appropriate. Your research has found that there is an official public image called "postgres" available on Docker Hub and another called "python" that can be used for running Python applications. Your team has made these images available from the local registry.

1. In the terminal on Ubuntu-Desktop, **type** `curl http://registry:5000/v2/_catalog` and **press Enter** to list the images available from the registry.



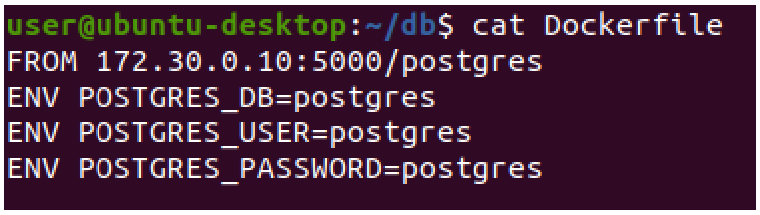
```
user@ubuntu-desktop:~/test$ curl http://registry:5000/v2/_catalog
{"repositories":["alpine","postgres","python"]}
```

Command to get the list of images from the registry

Note: You should see a JSON response with an array "repositories" containing "alpine," "postgres," and "python." These are the three base images that you need to complete your testing. You have already used the alpine image in the first part. For the first part of the application, you need to create a container running the database.

2. In the terminal, **type** `cd ~/db` and **press Enter** to change the working directory.

3. In the terminal, **type** `cat Dockerfile` and **press Enter** to view the Dockerfile.



```
user@ubuntu-desktop:~/db$ cat Dockerfile
FROM 172.30.0.10:5000/postgres
ENV POSTGRES_DB=postgres
ENV POSTGRES_USER=postgres
ENV POSTGRES_PASSWORD=postgres
```

Command to view the Dockerfile

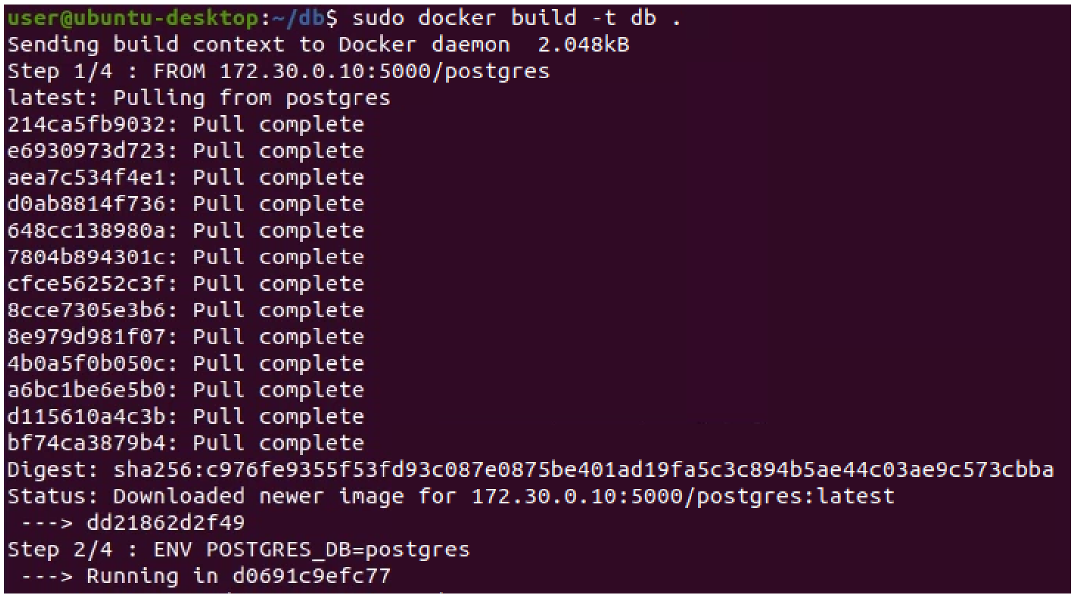
Note: The initial testing Dockerfile is minimal. The postgres image can make use of environment variables to specify the database and credentials. You can read about the options for this image here. In the Dockerfile, the FROM line indicates that the base image is postgres from the local registry. The ENV lines set environment variables the postgres image is designed to use.

- POSTGRES_DB is the name for the database.
- POSTGRES_USER is the username for database access.

- POSTGRES_PASSWORD is the password for database access.

You can use the build command to create the image and tag it as "db."

4. In the terminal, **type** `sudo docker build -t db .` and **press Enter** to create the image and tag it.

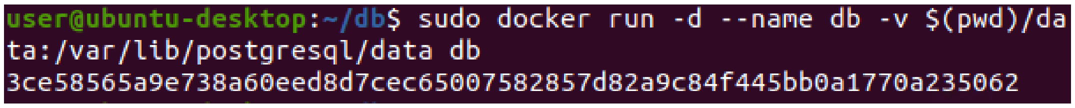


```
user@ubuntu-desktop:~/db$ sudo docker build -t db .
Sending build context to Docker daemon  2.048kB
Step 1/4 : FROM 172.30.0.10:5000/postgres
latest: Pulling from postgres
214ca5fb9032: Pull complete
e6930973d723: Pull complete
aea7c534f4e1: Pull complete
d0ab8814f736: Pull complete
648cc138980a: Pull complete
7804b894301c: Pull complete
cfce56252c3f: Pull complete
8cce7305e3b6: Pull complete
8e979d981f07: Pull complete
4b0a5f0b050c: Pull complete
a6bc1be6e5b0: Pull complete
d115610a4c3b: Pull complete
bf74ca3879b4: Pull complete
Digest: sha256:c976fe9355f53fd93c087e0875be401ad19fa5c3c894b5ae44c03ae9c573cbba
Status: Downloaded newer image for 172.30.0.10:5000/postgres:latest
 ---> dd21862d2f49
Step 2/4 : ENV POSTGRES_DB=postgres
 ---> Running in d0691c9efc77
```

Command to build the image

Note: You should see four steps in the build process. First, the image is pulled from the registry. Then each of the environment variables is set. Now run the image.

5. In the terminal, **type** `sudo docker run -d --name db -v $(pwd)/data:/var/lib/postgresql/data db` and **press Enter** to run the database image.



```
user@ubuntu-desktop:~/db$ sudo docker run -d --name db -v $(pwd)/da
ta:/var/lib/postgresql/data db
3ce58565a9e738a60eed8d7cec65007582857d82a9c84f445bb0a1770a235062
```

Comamnd to create and run the container

Note: This time, you used the shortcut version and did not manually create a container first. There are a few new arguments as well.

- -v $(pwd)/data:/var/lib/postgresql/data
  This tells Docker to use the folder data in the current working directory as /var/lib/postgresql/data in the container. The connection is "live" so that changes made by the container are persistent and visible from the host.

- -d
  Makes the container run detached and keeps running until you decide to stop it.

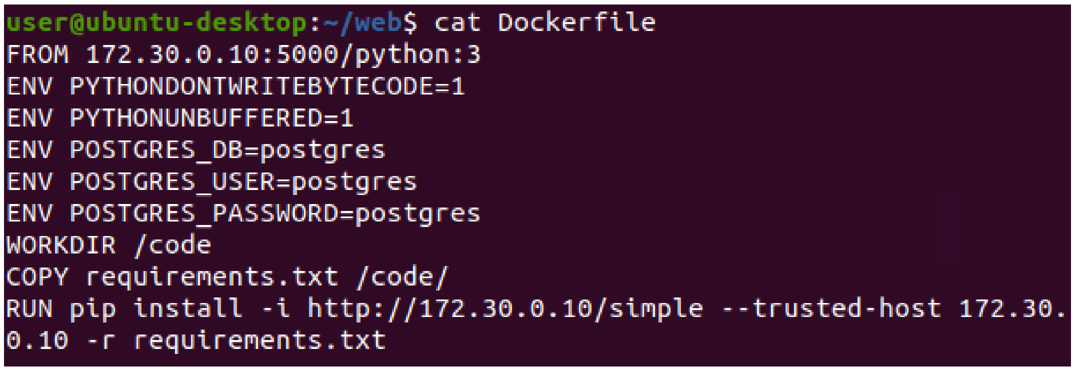
6. In the terminal, **type** `sudo docker ps` and **press Enter** to list the running containers.

7. **Make a screen capture** showing **the running db container**.

Note: Now you need to build the image for the web application. For testing, you will just use the generic Python image and initialize a Django example application.

8. In the terminal, **type** `cd ~/web` and **press Enter** to change the working directory.

9. In the terminal, **type** `cat Dockerfile` and **press Enter** to view the Dockerfile contents.



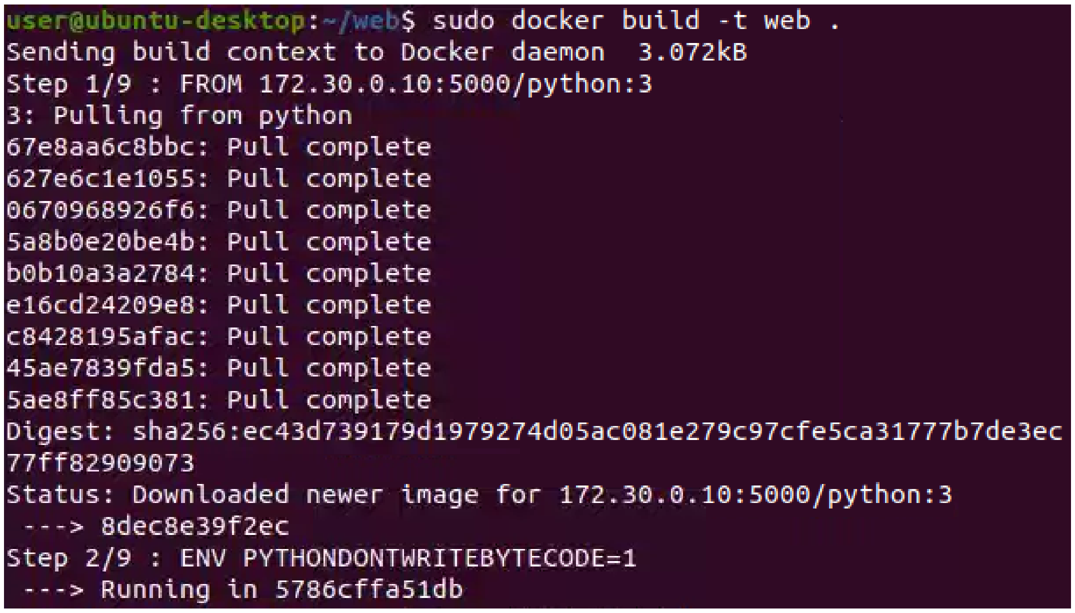
```
user@ubuntu-desktop:~/web$ cat Dockerfile
FROM 172.30.0.10:5000/python:3
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1
ENV POSTGRES_DB=postgres
ENV POSTGRES_USER=postgres
ENV POSTGRES_PASSWORD=postgres
WORKDIR /code
COPY requirements.txt /code/
RUN pip install -i http://172.30.0.10/simple --trusted-host 172.30.
0.10 -r requirements.txt
```

Command to view the Dockerfile

Note: This starting Python image is documented here. The Dockerfile includes some environment variables for controlling how Python behaves as well as the information about the database. It also includes a RUN line to actually install the Python packages listed in the requirements.txt file. Note that the requirements file must first be copied to the container before it can be used in the RUN command.

Another member of your team has set up the Docker-Server machine to also provide a local repository for Python packages. This will enable your company to maintain better control over the source for your containers.

10. In the terminal, **type** **`sudo docker build -t web .`** and **press Enter** to build the Docker image.

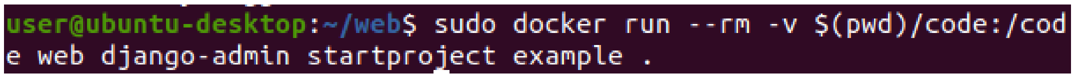


```
user@ubuntu-desktop:~/web$ sudo docker build -t web .
Sending build context to Docker daemon  3.072kB
Step 1/9 : FROM 172.30.0.10:5000/python:3
3: Pulling from python
67e8aa6c8bbc: Pull complete
627e6c1e1055: Pull complete
0670968926f6: Pull complete
5a8b0e20be4b: Pull complete
b0b10a3a2784: Pull complete
e16cd24209e8: Pull complete
c8428195afac: Pull complete
45ae7839fda5: Pull complete
5ae8ff85c381: Pull complete
Digest: sha256:ec43d739179d1979274d05ac081e279c97cfe5ca31777b7de3ec
77ff82909073
Status: Downloaded newer image for 172.30.0.10:5000/python:3
 ---> 8dec8e39f2ec
Step 2/9 : ENV PYTHONDONTWRITEBYTECODE=1
 ---> Running in 5786cffa51db
```

Command to create the image

Note: This Dockerfile has a few more steps. The first one retrieves the base Python image, which can take some time. Steps 2 through 8 should proceed quickly. The RUN step also takes time because it downloads and installs the required Python packages. In this step you will see a warning about running pip as root that you can ignore.

11. In the terminal, **type** `sudo docker run --rm -v $(pwd)/code:/code web django-admin startproject example .` and **press Enter** to initialize the Django app.



```
user@ubuntu-desktop:~/web$ sudo docker run --rm -v $(pwd)/code:/cod
e web django-admin startproject example .
```

Command to create and run the container and initialize the app

Note: The command does not have any output. For this command, you used the local ~/web/code directory and mounted it at /code in the container. Then the command "django-admin startproject example" was run in the container. This makes the code directory available on the host so that you can make changes that will be visible in the container. The new argument "--rm" means to remove the container after it exits.

12. In the terminal, **type** `sudo docker container ls --all` and **press Enter** to see the list of running containers.

Note: There is no container based on the web image listed. The previous command ran a command and then the container stopped when the command was done. Then the container was removed. The command that was run has initialized the code directory with a minimal Django app.

13. In the terminal, **type** `ls -al code` and **press Enter** to view the code directory files.

Note: By default root owns the created files. You can change the ownership to make the files editable by user. There is a settings file that needs to be updated.

14. In the terminal, **type** `sudo chown -R user:user code` and **press Enter** to change the ownership of all the files in the tree.

15. In the terminal, **type** `vim code/example/settings.py` and **press Enter** to open the app settings file in an editor.

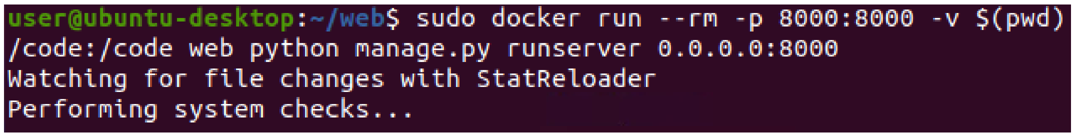
16. In the editor, **add the following line** after the line "from pathlib import Path":
    ```
    import os
    ```

17. In the editor, **update the databases section** to match the following, then press **ESC** and type `:wq!` to save and exit the editor:
    ```
    DATABASES = {
        'default': {
            'ENGINE': 'django.db.backends.postgresql',
            'NAME': os.environ.get('POSTGRES_NAME'),
            'USER': os.environ.get('POSTGRES_USER'),
            'PASSWORD': os.environ.get('POSTGRES_PASSWORD'),
    ```

```
            'HOST': 'db',
            'PORT': 5432,
        }
    }
```

Note: These changes tell Django how to connect to the PostgresQL database using the values of the environment variables that were defined. The next step is to run the web application container.

18. In the terminal, **type** `sudo docker run --rm -p 8000:8000 -v $(pwd)/code:/code web python manage.py runserver 0.0.0.0:8000` and **press Enter** to try to launch the application.



```
user@ubuntu-desktop:~/web$ sudo docker run --rm -p 8000:8000 -v $(pwd)
/code:/code web python manage.py runserver 0.0.0.0:8000
Watching for file changes with StatReloader
Performing system checks...
```

Command to create and run the web container

Note: Here you have the familiar arguments for volumes and removing the container when it is done. There is also a new one, -p 8000:8000. This tells the Docker daemon to connect port 8000 of the container to port 8000 of the host. With that connection, any attempts to connect to port 8000 on the host will be forwarded to the container. However, the command should result in an error ending with a line saying it could not translate host name "db" to an address.

The default bridge network for Docker containers does not support resolution based on names. The networking setup is part of the container definition. This is why --rm was included—the failed container has already been removed. However, the running db container needs to be on the same network, so you need to stop and remove that container.

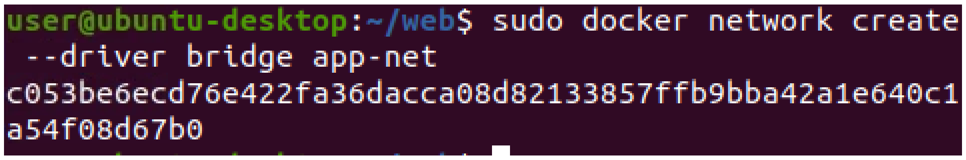
19. In the terminal, **press ctrl-c** to stop and remove the web container.

20. In the terminal, **type** `sudo docker container stop db` and **press Enter** to stop the db container.

21. In the terminal, **type** **`sudo docker container rm db`** and **press Enter** to remove the db container.

Note: To enable resolution by host name, you need to use a user-defined bridge network rather than the default network. You will need to use the "network create" command for this.

22. In the terminal, **type** **`sudo docker network create --driver bridge app-net`** and **press Enter** to create a new bridge network.



```
user@ubuntu-desktop:~/web$ sudo docker network create
 --driver bridge app-net
c053be6ecd76e422fa36dacca08d82133857ffb9bba42a1e640c1
a54f08d67b0
```

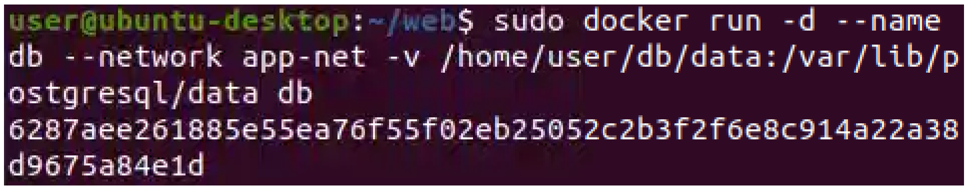
Command to create a network

23. In the terminal, **type** **`sudo docker network ls`** and **press Enter** to list the docker networks.

24. **Make a screen capture** showing the **command creating the network and the resulting id**.

Note: Now you can run containers with the database and web images connected to the new network.

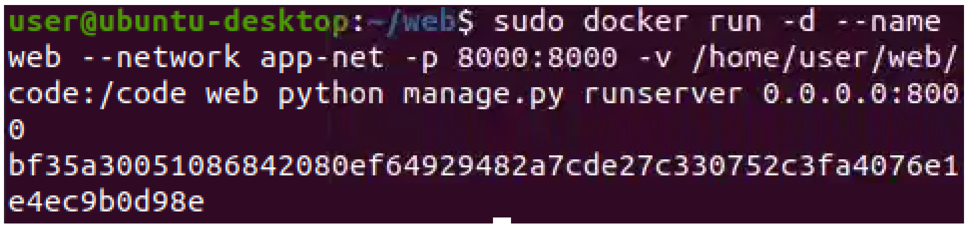
25. In the terminal, **type** **`sudo docker run -d --name db --network app-net -v /home/user/db/data:/var/lib/postgresql/data db`** and **press Enter** to run the database image in a container.



```
user@ubuntu-desktop:~/web$ sudo docker run -d --name
db --network app-net -v /home/user/db/data:/var/lib/p
ostgresql/data db
6287aee261885e55ea76f55f02eb25052c2b3f2f6e8c914a22a38
d9675a84e1d
```

Command to create and run the db container

26. In the terminal, **type** `sudo docker run -d --name web --network app-net -p 8000:8000 -v /home/user/web/code:/code web python manage.py runserver 0.0.0.0:8000` and **press Enter** to run the web application image in a container.



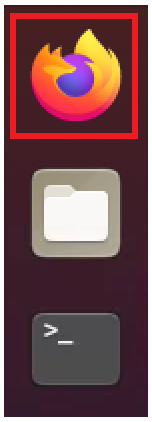
```
user@ubuntu-desktop:~/web$ sudo docker run -d --name
web --network app-net -p 8000:8000 -v /home/user/web/
code:/code web python manage.py runserver 0.0.0.0:800
0
bf35a30051086842080ef64929482a7cde27c330752c3fa4076e1
e4ec9b0d98e
```

Command to create and run the web container

27. In the terminal, **type** `sudo docker ps` and **press Enter** to view the running containers.

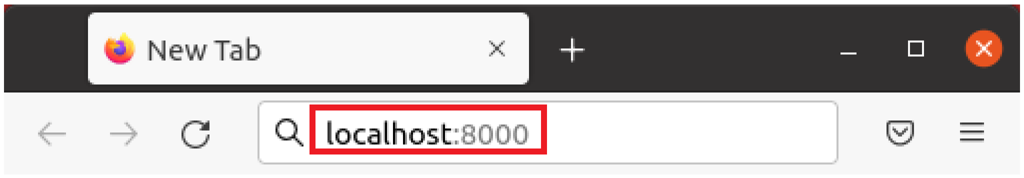
Note: You should see two running containers named web and db.

28. In the sidebar, **click** the **Firefox icon** to open the web browser.



Firefox Icon

29. In the browser, **navigate to localhost:8000**.



Navigate to localhost:8000

Note: You should see a page confirming that Django was successfully installed.

30. **Make a screen capture** showing **the Django page**.

Part 3: Using Docker Compose

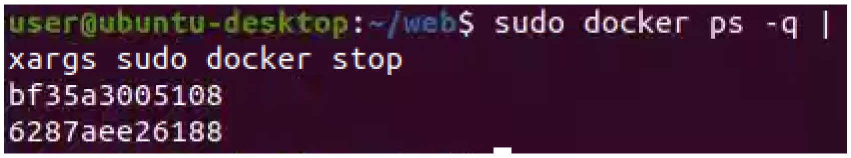
Note: The methods used for a multi-container application in Part 2 were to give a deeper understanding of how the parts all work together. It would be nice if the configuration could be specified and everything accomplished with fewer commands. This is, in fact, possible using Docker compose.
Docker Compose is controlled with a YAML configuration file with a default name of docker-

compose.yml in the current directory. This file has several possible top-level elements. A few of the important ones are

- services – describes the Docker containers that are a part of the application
- networks – describes networks to be created for the application
- volumes – describes volumes that can be used with multiple services

You can view the full documentation for the compose files here. For your testing, you want to keep using the same images from the previous part. To avoid any name overlaps, first stop and remove any containers that exist.

1. In the terminal, **type** `sudo docker ps -q | xargs sudo docker stop` and **press Enter** to stop all running containers.



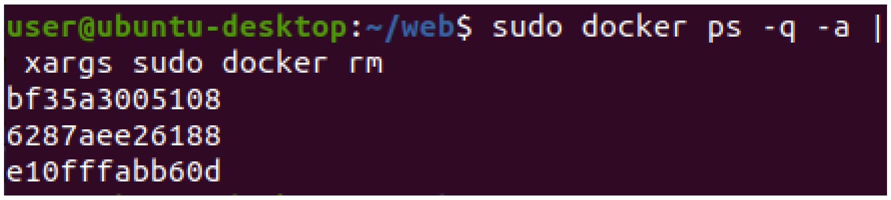
```
user@ubuntu-desktop:~/web$ sudo docker ps -q |
xargs sudo docker stop
bf35a3005108
6287aee26188
```

Command to stop all containers

Note: The "-q" option to the docker ps command makes it list container IDs one per line. The xargs command is a standard POSIX tool for running multiple instances of a command. By default it makes each command run by appending an input line to the end of the rest of the arguments. In this case, the command will run "sudo docker stop CONTAINER_ID" for each ID that is generated by the docker ps command.

Alternatively, you could use command substitution in the same vein as your earlier directory substitutions with $(pwd). For example, sudo docker stop $(docker ps -q -a) and sudo docker rm $(docker ps -q -a).

2. In the terminal, **type** `sudo docker ps -q -a | xargs sudo docker rm` and **press Enter** to remove all the Docker containers.



```
user@ubuntu-desktop:~/web$ sudo docker ps -q -a |
 xargs sudo docker rm
bf35a3005108
6287aee26188
e10fffabb60d
```

Command to remove all containers

Note: The -a argument for docker ps makes it list all containers, not just running ones. This allows the xargs command to remove all containers. Now you can clean up the files generated in the previous part to make sure the Docker Compose process is complete.

3. In the terminal, **type** `cp ~/web/code/example/settings.py ~/.` and **press Enter** to copy the settings file for reuse.

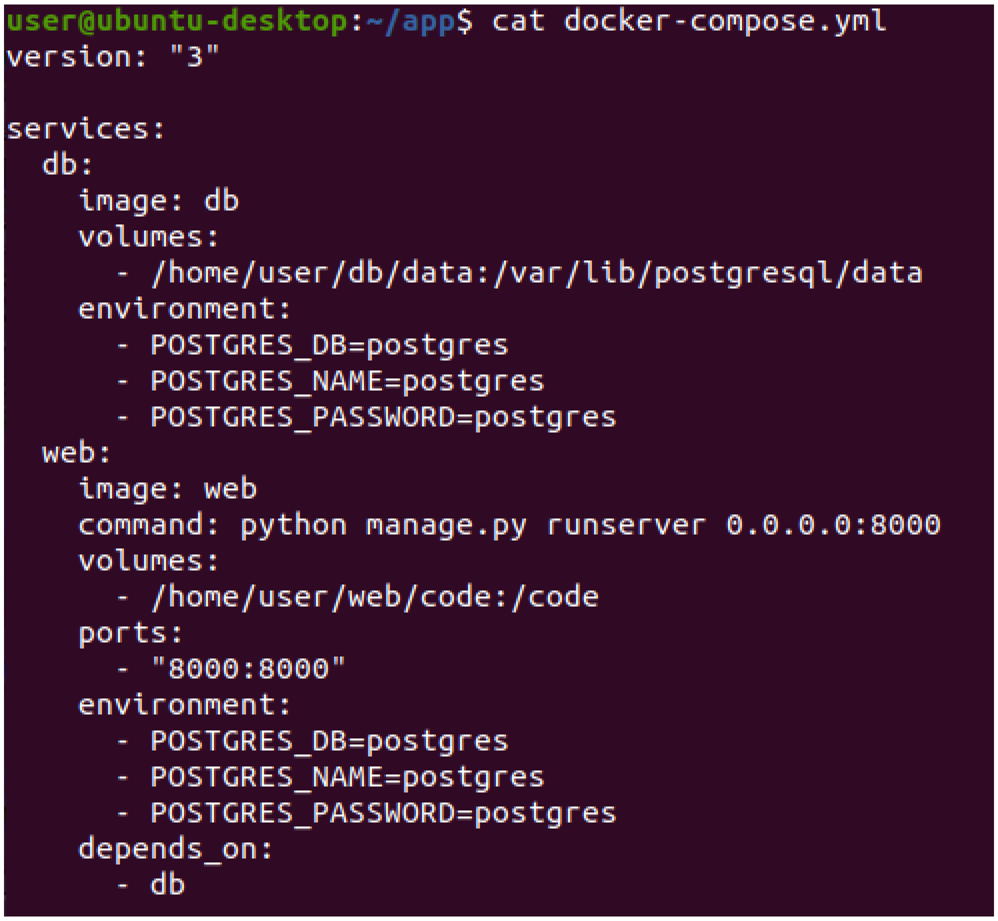
4. In the terminal, **type** `sudo rm -rf ~/web/code` and **press Enter** to remove the application files created in Part 2.

5. In the terminal, **type** `sudo rm -rf ~/db/data` and **press Enter** to remove the database files from Part 2.

Note: In your initial research, you have created a docker-compose.yml file to be used.

6. In the terminal, **type** `cd ~/app` and **press Enter** to change the working directory.

7. In the terminal, **type** `cat docker-compose.yml` and **press Enter** to view the YAML file.



```
user@ubuntu-desktop:~/app$ cat docker-compose.yml
version: "3"

services:
  db:
    image: db
    volumes:
      - /home/user/db/data:/var/lib/postgresql/data
    environment:
      - POSTGRES_DB=postgres
      - POSTGRES_NAME=postgres
      - POSTGRES_PASSWORD=postgres
  web:
    image: web
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - /home/user/web/code:/code
    ports:
      - "8000:8000"
    environment:
      - POSTGRES_DB=postgres
      - POSTGRES_NAME=postgres
      - POSTGRES_PASSWORD=postgres
    depends_on:
      - db
```

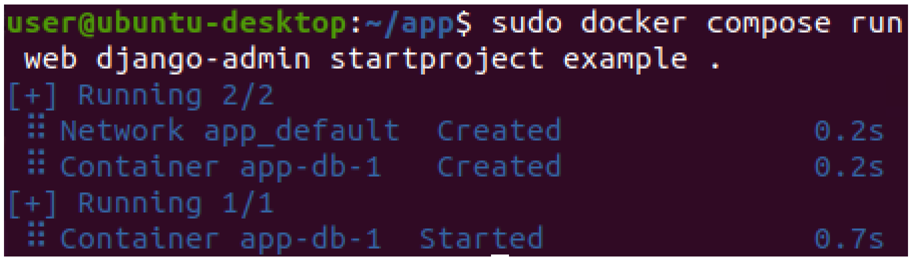
Command to view the Docker compose file

Note: Each service corresponds to a container you want to have running for the application. The web and db images have already been created and can be reused. Your application requires two containers: db and web. Each one has an image property naming the image to be used for running the container. Each one also has environment sections that specify environment variables for the running container. There is also a section for volumes in each service. These contain both the path on the host and the path in the container. As you can see, these parameters contain the information that was provided in some of the commands used in the previous section.

The postgres image (and by extension the db image) has a default command that starts the database service and keeps the container running. It does not need a separate command. The web service has a command property that specifies the command to run for the container. The web service also has a property that indicates that it depends on the db service. The Docker Compose command will ensure that the db service is started first.

The application requires some manual setup. Docker Compose provides a run command that can be used for this purpose, similar to the "docker run" commands.

8. In the terminal, **type** `sudo docker compose run web django-admin startproject example .` and **press Enter** to initialize the Django application.



```
user@ubuntu-desktop:~/app$ sudo docker compose run
 web django-admin startproject example .
[+] Running 2/2
 ⠿ Network app_default  Created               0.2s
 ⠿ Container app-db-1   Created               0.2s
[+] Running 1/1
 ⠿ Container app-db-1  Started                0.7s
```

Command to initialize the web app

Note: This command will create the /home/user/web/code directory again. You can copy the saved settings file to the appropriate location.

9. In the terminal, **type** `sudo cp ~/settings.py ~/web/code/example/settings.py` and **press Enter** to replace the settings file for the application.

10. In the terminal, **type** `sudo docker compose up` and **press Enter** to start all the containers needed for the application.

11. **Make a screen capture** showing **output from the compose up command**.

12. In the web browser, **navigate to localhost:8000** to confirm that the application is running.

13. **Restore** the terminal and **press ctrl-c** to stop and remove the web container.

Challenge and Analysis

Note: The following scenario is provided to allow independent, unguided work – similar to what you will encounter in a real-world situation.

Part 1: Create an Image to Serve a Static File

Some of your company's applications include simple static files to be served. You would like to create separate containers to serve static content. For testing, you can use the Python http server in the standard library, which can be run using "python -m http.server -d <path> <port>" to serve the files in <path> listening on the given port.

Create a new folder /home/user/web2 with a Dockerfile and a subfolder www containing a static file to be served. Write the Dockerfile so that it includes the default Python http server command to run for the web server, using port 8001. Once you have assembled the necessary steps in your Dockerfile, execute the command to build it.

Make a screen capture showing the **successful build of the web2 image.**

Now that your image is built, create and run a container based on it. You may do this in several steps, or use the appropriate docker command to accomplish the task in a single step. Ensure that you connect port 8001 of the docker container to port 8001 on the host. Also ensure that you map the www subfolder you created to the directory in the container that you are serving your static file from.

In Firefox, navigate to localhost:8001 to show the directory listing on port 8001, which should contain your static file.

Make a screen capture showing the **directory listing served on port 8001**.

Part 2: Add the Static Service to the Docker Compose File

Any static web services should be started as a part of the multi-container application. Update the file docker-compose.yml of your company's Django-based application to use the new web2 image created in the previous part. This new service should be named web2 and should be exposed on port 8080 of the Docker host.

Once you have updated the docker-compose.yml file, use docker compose to start the multi-container Django application. The output of your command should indicate all three containers (db, web, and web2) were created.

Make a screen capture showing the **creation of the web, web2, and db containers** in the docker compose output.

In Firefox, navigate to your web2 service running on port 8080 to display a directory listing that contains the file you created in the previous part.

Make a screen capture showing the **directory listing served on port 8080**.