Creating and Deploying a Sample Application

6.2.1

What is Docker?

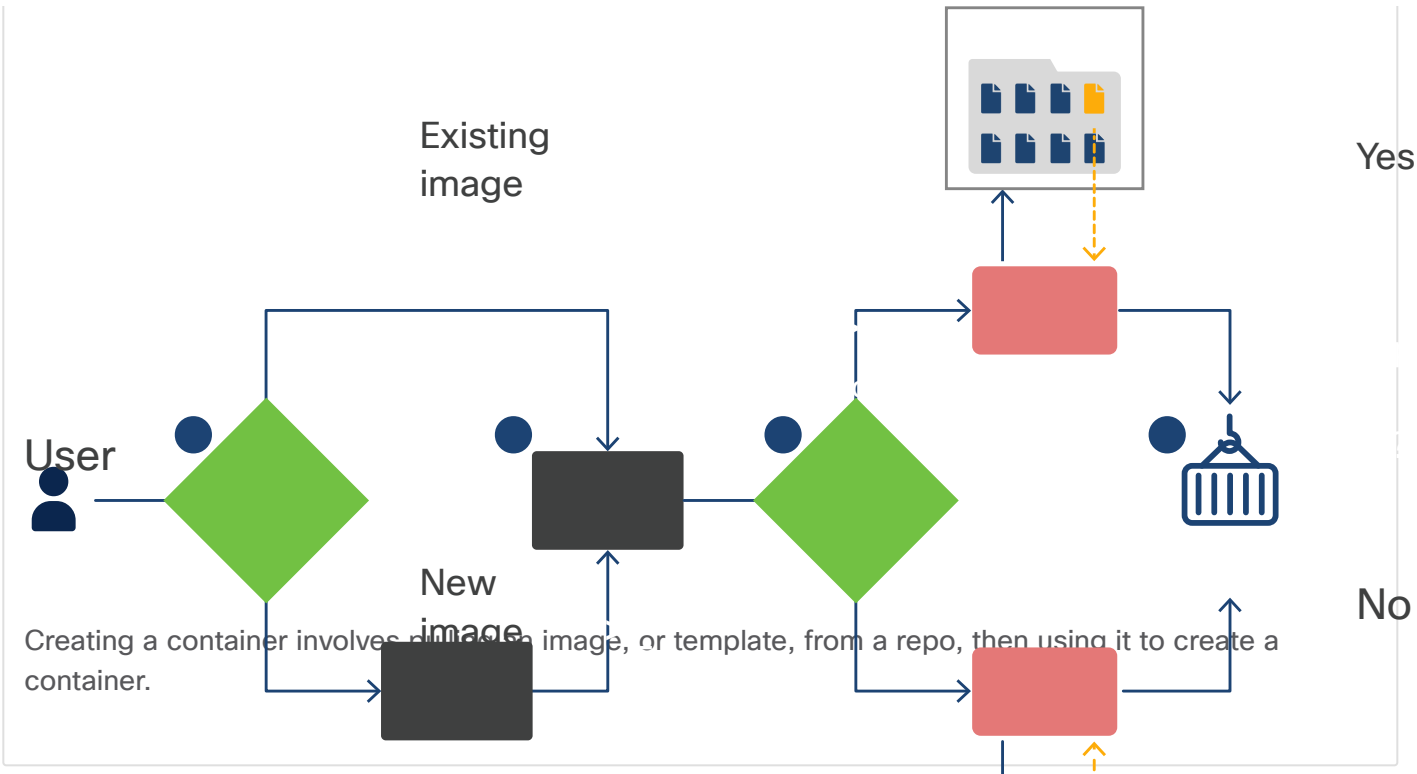
The most popular way to containerize an application is to deploy it as a Docker container. A container is a way of encapsulating everything you need to run your application, so that it can easily be deployed in a variety of environments. Docker is a way of creating and running that container. Specifically, Docker is a format that wraps a number of different technologies to create what we know today as containers. These technologies are:

- **Namespaces** - These isolate different parts of the running container. For example, the process itself is isolated in the pid (process ID) namespace, the filesystem is isolated in the `mnt` (mount) namespace, and networking is isolated in the net namespace.
- **Control groups** - These `cgroups` are a standard linux concept that enables the system to limit the resources, such as RAM or storage, used by an application.
- **Union File Systems** - These `UnionFS` are file systems that are built layer by layer, combining resources.

A Docker image is a set of read-only files which has no state. A Docker Image contains source code, libraries, and other dependencies needed to run an application. A Docker container is the run-time instance of a Docker image. You can have many running containers of the same Docker image. A Docker image is like a recipe for a cake, and you can make as many cakes (Docker containers) as you wish.

Images can in turn be stored in registries such as Docker Hub. Overall, the system looks like this:

Request i



So a simplified version of the workflow of creating a container looks like t

Step 1. Either create a new image using `docker build` or pull a copy o age from a registry using docker pull. (Depending on the circumstances, this step is optional

Step 2. Run a container based on the image using `docker run` or `docker container create`.

Step 3. The Docker daemon checks to see if it has a local copy of the image. If it does not, it pulls the image from the registry.

Step 4. The Docker daemon creates a container based on the image and, if `docker run` was used, logs into it and executes the requested command.

As you can see, if you are going to create a container-based deployment of the sample application, you are going to have to create an image. To do that, you need a `Dockerfile`.

6.2.2

What is a Dockerfile?

If you have used a coding language such as C, you know that it required you to compile your code. If so, you may be familiar with the concept of a “makefile.” This is the file that the make utility uses to compile and build all the pieces of the application.

That is what a `Dockerfile` does for Docker. It is a simple text file, named `Dockerfile`. It defines the steps that the `docker build` command needs to take to create an image that can then be used to create the target container.

You can create a very simple `Dockerfile` that creates an Ubuntu container. Use the cat command to create a `Dockerfile`, and then add `FROM ubuntu` to the file. Enter `Ctrl+D` to save and exit the file with the following text and save it in your current directory:

```
devasc@labvm:~$ cat > Dockerfile
FROM ubuntu:latest
<Ctrl+D>
devasc@labvm:~$
```

That is all it takes, just that one line. Now you can use the `docker build` command to build the image as shown in the following example. The `-t` option is used to name the build. Notice the period (`.`) at the end of the command which specifies that the image should be built in the current directory. Use `docker build --help` to see all the available options.

```
devasc@labvm:~$ docker build -t myubuntu:latest .
Sending build context to Docker daemon  983.3MB
Step 1/1 : FROM ubuntu:latest
latest: Pulling from library/ubuntu
692c352adcf2: Pull complete
97058a342707: Pull complete
2821b8e766f4: Pull complete
4e643cc37772: Pull complete
Digest: sha256:55cd38b70425947db71112eb5dddfa3aa3e3ce307754a3df2269069d2278ce47
Status: Downloaded newer image for ubuntu:latest
 ---> adafef2e596e
Successfully built adafef2e596e
Successfully tagged myubuntu:latest
devasc@labvm:~$
```

Enter the command `docker images` to see your image in the list of images on the DEVASC VM:

```
devasc@labvm:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
myubuntu            latest              adafef2e596e        3 days ago          73.9MB
ubuntu              latest              adafef2e596e        3 days ago          73.9MB
devasc@labvm:~$
```

Now that you have the image, use the `docker run` command to run it. You are now in a bash shell INSIDE the docker image you created. Change to the home directory and enter `ls` to see that it is empty and ready for use. Enter `exit` to leave the Docker container and return to your DEVASC VM main operating system.

```
devasc@labvm:~$ docker run -it myubuntu:latest /bin/sh
# ls
bin  boot  dev  etc  home  lib  lib32  lib64  libx32  media  mnt  opt  proc  root  run
sbin  srv  sys  tmp  usr  var
# cd home
# ls
# exit
devasc@labvm:~$
```

6.2.3

Anatomy of a Dockerfile

Of course, if all you could do with a `Dockerfile` was to start a clean operating system, that would be useful, but what you need is a way to start with a template and build from there.

Note: The steps shown in this rest of this topic are for instruction purposes only. Additional details that you would need to complete these commands in your DEVASC VM are not provided. However, you will complete similar steps in the lab *Build a Sample Web App in a Docker Container* later in the topic.

Consider the following `Dockerfile` that containerizes a Python app:

```
FROM python
WORKDIR /home/ubuntu
COPY ./sample-app.py /home/ubuntu/.
RUN pip install flask
CMD python /home/ubuntu/sample-app.py
EXPOSE 8080
```

In the `Dockerfile` above, an explanation of the commands are as follows:

- The `FROM` command installs Python in the Docker image. It invokes a Debian Linux-based default image from Docker Hub, with the latest version of Python installed.
- The `WORKDIR` command tells Docker to use `/home/ubuntu` as the working directory.
- The `COPY` command tells Docker to copy the **sample-app.py** file from Dockerfile's current directory into `/home/ubuntu`.
- The `RUN` command allows you to directly run commands on the container. In this example, Flask is installed. Flask is a platform to support your app as a web app.
- The `CMD` command will start the server when you run the actual container. Here, you use the python command to run the **sample-app.py** inside the container.
- The `EXPOSE` command tells Docker that you want to expose port 8080. Note that this is the port on which Flask is listening. If you have configured your web server to listen somewhere else (such as https

requests on port 443) this is the place to note it.

Use the `docker build` command to build the image. In the following output, the image was previously built. Therefore, Docker takes advantage of what is stored in cache to speed up the process.

```
$ docker build -t sample-app-image .
Sending build context to Docker daemon  3.072kB
Step 1/6 : FROM python
 ---> 0a3a95c81a2b
Step 2/6 : WORKDIR /home/ubuntu
 ---> Using cache
 ---> 17befcf89bab
Step 3/6 : COPY ./sample-app.py /home/ubuntu/.
 ---> Using cache
 ---> c0b3a4f9c568
Step 4/6 : RUN pip install flask
 ---> Using cache
 ---> 8cf8226c9f31
Step 5/6 : CMD python /home/ubuntu/sample-app.py
 ---> Running in 267c5d569356
Removing intermediate container 267c5d569356
 ---> 75cd4bf1d02a
Step 6/6 : EXPOSE 8080
 ---> Running in cc82eaca2028
Removing intermediate container cc82eaca2028
 ---> 9616439582f8
Successfully built 9616439582f8
Successfully tagged sample-app-image:latest
$
```

As you can see, Docker goes through each step in the Dockerfile, starting with the base image, Python. If this image does not exist on your system, Docker pulls it from the registry. The default registry is Docker Hub. However, in a secure environment, you might set up your own registry of trusted container images. Notice that the image is actually a number of different images layered on top of each other, just as you are layering your own commands on top of the base image.

Notice that between steps such as executing a command, Docker actually creates a new container and builds an intermediate image, a new layer, by saving that container. In fact, you can do that yourself by creating a container, making the changes you want, then saving that container as a new image.

In the previous example, only a small number of the available Dockerfile commands were used. The complete list is available in the Docker documentation in the Dockerfile reference. Currently a list of available commands looks like this:

- FROM
- MAINTAINER
- RUN

- CMD
- EXPOSE
- ENV
- COPY
- ENTRYPOINT
- VOLUME
- USER
- WORKDIR
- ARG
- ONBUILD
- STOPSIGNAL
- LABEL

Enter the command `docker images` to view a list of images. Notice that there are actually two images that are now cached on the machine. The first is the Python image, which you used as your base. Docker has stored it so that if you were to rebuild your image, you will not have to download it again.

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
sample-app-image    latest              7b1fd666ae4c        About an hour ago   410MB
python              latest              daddc1037fdf        2 days ago          410MB
$
```

6.2.4

Start a Docker Container Locally

Now that image is created, use it to create a new container and actually do some work by entering the `docker run` command, as shown in the following output. In this case, several parameters are specified. The `-d` parameter is short for `--detach` and says you want to run it in the background. The `-P` parameter tells Docker to publish it on the ports that you exposed (in this case, `8080`).

```
$ docker run -d -P sample-app-image
1688a2c34c9e7725c38e3d9262117f1124f54685841e97c3c5225af88e30bfc5
$
```

You can see the container by listing processes:

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                    NAMES
90edd03a9511        sample-app-image    "/bin/sh -c 'python …"   5 seconds ago
```

```
Up 3 seconds        0.0.0.0:32774->8080/tcp   jovial_sammet
$
```

There are a few things to note here. Working backwards, notice that Docker has assigned the container a name, `jovial_sammet`. You could also have named it yourself with the `--name` option. For example:

```
docker run -d -P --name pythontest sample-app-image
```

Notice also that, even though the container is listening on port 8080, that is just an internal port. Docker has specified an external port, in this case 32774, that will forward to that internal port. This lets you run multiple containers that listen on the same port without having conflicts. If you want to pull up your sample app website, you can use the public IP address for the host server and that port. Alternatively, if you were to call it from the host machine itself, you would still use that externalized port, as shown with the following curl command.

```
$ curl localhost:32774
You are calling me from 172.17.0.1
$
```

Docker also lets you specify a particular port to forward, so that you can create a more predictable system:

```
$ docker run -d -p 8080:8080 --name pythontest sample-app-image
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                    NAMES
a51da037bf35        sample-app-image    "/bin/sh -c 'python …"   28 seconds ago
Up 27 seconds       0.0.0.0:8080->8080/tcp    pythontest
90edd03a9511        sample-app-image    "/bin/sh -c 'python …"   24 minutes ago
Up 24 minutes       0.0.0.0:32774->8080/tcp   jovial_sammet
$
```

When your container is running, you can log into it just as you would log into any physical or virtual host using the `exec` command from the host on which the container is running:

```
$ docker exec -it pythontest /bin/sh
# whoami
root
# pwd
/var/www/html
# exit
$
```

To stop and remove a running container, you can call it by its name:

```
$ docker stop pythontest
pythontest
$ docker rm pythontest
pythontest
$
```

Now if you look at the running processes again, you can see that it is gone.

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                    NAMES
90edd03a9511        sample-app-image    "/bin/sh -c 'python …"   25 minutes ago
Up 25 minutes       0.0.0.0:32774->8080/tcp   jovial_sammet
$
```

6.2.5

Save a Docker Image to a Registry

Now that you know how to create and use your image, it is time to make it available for other people to use. One way to do this is by storing it in an image registry.

By default, Docker uses the Docker Hub registry, though you can create and use your own registry. You will need to start by logging in to the registry:

```
$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a
Docker ID, head over to https://hub.docker.com to create one.
Username: devnetstudent # This would be your username
Password:               # This would be your password
WARNING! Your password will be stored unencrypted in /home/ubuntu/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store
Login Succeeded
$
```

Next, you commit a running container instance of your image. For example, the `pythontest` container is running in this example. Commit the container with the `docker commit` command.

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

```
54c44606344c        sample-app-image    "/bin/sh -c 'python …"   4 seconds ago
Up 2 seconds        0.0.0.0:8080->8080/tcp    pythontest
$ docker commit pythontest sample-app
Sha256:bddc326383032598a1c1c2916ce5a944849d90e4db0a34b139eb315af266e68b
$
```

Next, use the `docker tag` command to give the image you commited a tag. The tag takes the following form:

`<repository>/<imagename>:<tag>`

The first part, the repository, is usually the username of the account storing the image. In this example, it is **devnetstudent**. Next is the image name, and then finally the optional tag. (Remember, if you do not specify it, it will come up as latest.)

In this example, the tag could be **v1**, as shown here:

```
$ docker tag sample-app devnetstudent/sample-app:v1
$
```

Now the image is ready to be pushed to the repository:

```
$ docker push devnetstudent/sample-app:v1
The push refers to repository [docker.io/nickchase/sample-app]
e842dba90a43: Pushed
868914f88a69: Pushed
c7d71f6230b3: Pushed
1ed9b15dd229: Pushed
00947a3aa859: Mounted from library/python
7290ddeeb6e8: Mounted from library/python
d3bfe2faf397: Mounted from library/python
cecea5b3282e: Mounted from library/python
9437609235f0: Mounted from library/python
bee1c15bf7e8: Mounted from library/python
423d63eb4a27: Mounted from library/python
7f9bf938b053: Mounted from library/python
f2b4f0674ba3: Mounted from library/python
v1: digest: sha256:28e119f43e9c8e5e44f167d9baf113cc91d4f8b461714cd6bb578ebb0654f243
size: 3052
$
```

From here you can see that the new image is stored locally:

```
$ docker images
REPOSITORY            TAG                 IMAGE ID            CREATED
SIZE
```

```
sample-app              latest              bddc32638303        About a minute ago
410MB
devnetstudent/sample-app   v1                  bddc32638303        About a minute ago
410MB
$
```

6.2.6

Create a Development Environment

As you may recall, there are four different environments in a typical workflow:

- The Development environment
- The Testing environment
- The Staging environment
- The Production environment

Start by creating the development environment.

The development environment is meant to be convenient to the developer; it only needs to match the production environment where it is relevant. For example, if the developer is working on functionality that has nothing to do with the database, the development environment does not need a replica of the production database, or any database at all.

A typical development environment can consist of any number of tools, from Integrated Development Environments (IDEs) such as Eclipse to databases to object storage. The important part here is that it has to be comfortable for the developer.

In this case, you are going to build a simple Python app with tools available from the basic command line, Bash. You can also use Bash to perform testing and deployment tasks, so start with a Bash refresher.

6.2.7

Lab – Build a Sample Web App in a Docker Container

In this lab, you will review basic bash scripting techniques because bash scripting is a prerequisite for the rest of the lab. You will then build and modify a Python script for a simple web application. Next, you will create a bash script to automate the process for creating a Dockerfile, building the Docker container, and running the Docker container. Finally, you will use docker commands to investigate the intricacies of the Docker container instance.

You will complete the following objectives:

- Part 1: Launch the DEVASC VM
- Part 2: Create a Simple Bash Script
- Part 3: Create a Sample Web App
- Part 4: Configure the Web App to Use Website Files
- Part 5: Create a Bash Script to Build and Run a Docker Container
- Part 6: Build, Run, and Verify the Docker Container

Build a Sample Web App in a Docker Container

6.1
Understanding Deployment Choices with Di...

6.3
Continuous Integration/Continuous Deploym...