

Practice Docker Basics

Objective: In this lesson we will attempt to fill in some of the holes from the last lesson, highlight some other helpful commands when dealing with containers, and look at what to do when we are done using our container, or image.

Preparation: Prepare by opening up two terminal windows. One to serve as the primary and the other as a secondary. Also, navigate to the appropriate lab folder, where the lab data/material is.

Outcome: This lesson will give the participant a more thorough overview of Docker. By the end, we'll have started two separate containers, created a file in one, start and stop our container, and then remove it all when we're done.

Data Files: `hello.py`, `Dockerfile`

Learning a new technology can be very overwhelming. Docker does a great job though of indexing with their help menus. In this lab we will learn more about the lifecycle of containers, how to stop and start them, execute commands inside them, how to remove them, and how to build an image from a Dockerfile.

Step 1. Learn Container Run Options

As we discussed in the last lesson, ("Introduction to Docker Containers")

our containers don't necessarily stay running, but what else can we do when we run a container?

1. In your terminal, type `docker run --help` and let's look at a list of run options.

*NOTE: The list is too large to print here. Consult the instructor if you think you aren't getting the appropriate output.
*

Here's a list of some of the ones we will cover in this course:

<code>-d, --detach</code>	Run container in background and print container ID
<code>-e, --env list</code>	Set environment variables (default [])
<code>-h, --hostname string</code>	Container host name
<code>-i, --interactive</code>	Keep STDIN open even if not attached
<code>--name string</code>	Assign a name to the container
<code>--network string</code>	Connect a container to a network (default "default")
<code>-t, --tty</code>	Allocate a pseudo-TTY
<code>-v, --volume list</code>	Bind mount a volume (default [])

2. Two of the most used options, we normally see put together. The options (`-i`) and (`-t`) can be put together (`-it`) when running a container to make it interactive. And we can use the (`--name`) option to add a unique name identifier. Try it out, like this:

```
$ docker run -it --name Bert ubuntu:14.04
```

Notice that you are now inside the running container! Run (`pwd`) to see where you are inside the container.

3. Using the secondary terminal window, check to see if your container is indeed running. Can you remember the command? (Look back at “Introduction to Docker Containers” if you can not.)
4. Now, using that same secondary terminal window, launch another container in daemon mode by using the (`-d`) option. We will give it a name like we did previously, but this time we will also command (`echo`), argument (`hey-there`), as follows:

```
$ docker run -d --name Ernie ubuntu:14.04 echo hey-there
```

5. It's important that you understand what just happened. You can see it by listing your running containers, but this time you're gonna use (`-a`) to see all containers:

```
$ docker ps -a
```

Notice that only Bert is still running, the interactive (`-it`) one. Ernie, the one we assigned (`-d`), appears to have been exited. This is because Ernie was run in detach mode and given a specific command. Ernie simply started up, execute it's command, and exited out.

Step 2. Explore and Create

For this step, leave the secondary terminal window and return to the primary. If that container is not currently active, meaning you are not still logged into it, repeat point #2 of Step 1, or consult with the instructor.

1. Take a minute and explore the interactive Ubuntu container we've spun up. Once done, following along further to insure our container is up to date.

```
# sudo apt-get update
```

NOTE: We will be using (#) instead of (\$), because we are working inside a container.

2. Now, let's create a file. We will name it `hello.py` and you can fill it up with just about anything you want to. Following along here, and make sure to put the file in your Home directory.

```
# cd ~  
# pwd //this is just to confirm you are in </root>  
# touch hello.py
```

3. Now that we've created our file, let's go in and fill it up with our code.

```
# vi hello.py
```

Input:

```
hello-world
```

Vim (**vi**) Basic Instruction:

- Before writing any text you must press (**i**). To save your text to the **hello.py** doc, hit (**esc**) and then type out (**:qw**) and hit (**enter**).

If this is ineffective consult with the instructor.

Step 3. Stop, Start, and Execute

1. Now, leave the container running in your primary terminal window and preform the following commands in your secondary window, to stop the latter container:

```
$ docker stop <container-id, or name>
```

NOTE: The names (-name) we assigned were Bert and Ernie, and we know Ernie is down at this moment. So, Bert is the container name to use.

2. Start the container back up now and confirm that **hello.py** is still available:

```
$ docker start <container-id, or name>
```

*NOTE: This is when the (**--name**) option flag is best served. You can also use the first 2-4 numbers of the container-id. Docker will pick up*

on it.

3. Execute a command inside the container, concerning our `hello.py` file, using the execute (`exec`) command. Before we run our command though, let's look at the syntax of the (`exec`) command by typing into our terminal `docker exec` . Now, let's run ours:

```
$ docker exec <container-id, or name> tail /root/hello.py
```

4. What if we wanna go back into our container? Well, we can use the execute (`exec`) command for that as well, to invoke our bash shell, as follows:

```
$ docker exec -it <container-id, or name> bash  
exit
```

Step 4. Building Images With Dockerfile

We are going to add one more element here and put it all together now. Later in this lesson we will wipe it all away to start fresh. When we talk about Docker though it is important that we talk about Dockerfiles. Remember, Dockerfiles make images, images make containers. Let's get to work!

1. In your secondary window, download this lesson's Dockerfile or consult the instructor on where you can find it. Once you've secured your Dockerfile, use VIM (`vi`) to open it up into the editor. In case, you created a new file and copy pasted the content inside

that then you should give name as “Dockerfile”.

Notice the syntax:

```
# Comment  
INSTRUCTION arguments
```

Usage Definitions:

```
FROM      The base image to use in the build. This is mandatory  
           and must be the first command in the file.  
RUN       Executes a command and save the result as a new layer  
ADD       Copies a file from the host system onto the container  
WORKDIR   Set the default working directory for the container  
EXPOSE    Opens a port for linked containers
```

2. Let’s talk about the **build** command. Enact the help menu for build to review syntax and additional flag options:

```
$ docker build --help
```

Output:

```
docker build [OPTIONS] PATH | URL | -  
  
Options:  
    --build-arg list          Set build-time variables (
```

default [])

--cache-from stringSlice Images to consider as cache sources

--cgroup-parent string Optional parent cgroup for the container

--compress Compress the build context using gzip

--cpu-period int Limit the CPU CFS (Completely Fair Scheduler) period

--cpu-quota int Limit the CPU CFS (Completely Fair Scheduler) quota

-c, --cpu-shares int CPU shares (relative weight)

--cpuset-cpus string CPUs in which to allow execution (0-3, 0,1)

--cpuset-mems string MEMs in which to allow execution (0-3, 0,1)

--disable-content-trust Skip image verification (default true)

-f, --file string Name of the Dockerfile (Default is 'PATH/Dockerfile')

--force-rm Always remove intermediate containers

--help Print usage

--isolation string Container isolation technology

--label list Set metadata for an image (default [])

<code>-m, --memory string</code>	Memory limit
<code>--memory-swap string</code>	Swap limit equal to memory plus swap: '-1' to enable unlimited swap
<code>--network string</code>	Set the networking mode for the RUN instructions during build (default "default")
<code>--no-cache</code>	Do not use cache when building the image
<code>--pull</code>	Always attempt to pull a newer version of the image
<code>-q, --quiet</code>	Suppress the build output and print image ID on success
<code>--rm</code>	Remove intermediate containers after a successful build (default true)
<code>--security-opt stringSlice</code>	Security options
<code>--shm-size string</code>	Size of /dev/shm, default value is 64MB
<code>--squash</code>	Squash newly built layers into a single new layer
<code>-t, --tag list</code>	Name and optionally a tag in the 'name:tag' format (default [])
<code>--ulimit ulimit</code>	Ulimit options (default [])

3. Build an image from the Dockerfile now using the following command:

```
$ docker build -t hello-world-example .
```

NOTE: Ask your instructor if you don't know where to find your Dockerfile.

4. Once, the image is successfully built, we want to spin up a container from it, like this:

```
$ docker run hello-world-example
```

Output:

```
Traceback (most recent call last):
  File "hello.py", line 1, in <module>
    **hello-world**
NameError: name 'hello' is not defined
```

It ain't pretty, but it did run our `hello.py` file and print it's contents. From this you can begin to see how we might build an entire application like this.

Step 5. Removing Images and Containers

If any of our interactive (`-it`) container are still running, close it `CTL+C` . We also need to close any others down and remove all other previously used images from our local repository list so we can start a new project. We can do that by issuing a series or commands, as follows:

1. Print a list of all currently running and exited containers:

```
$ docker ps -a
```

2. Remove containers from the list, using:

```
$ docker rm <container-id, or name>
```

Because our container is currently active, we'll need to force (**-f**) the removal process. We could also use the (**kill**) command, before removing, but we'll use force this time.

```
$ docker rm -f <container-id, or name>
```

NTOE: You don't want to make a practice of deleting all your instances, because it will cost you time. Each image you delete must be downloaded, or built again. This is done for learning purposes only.

3. Print a list of current local repository images:

```
$ docker images
```

Try adding all(**-a**) and then compare the lists together.

4. Remove images now from local repository of images:

```
$ docker rmi <image-id, or name-type>
```

Conclusion

In this lesson we ran multiple run options, the execute command, and learned to remove both containers and images. We used (`-it`) to spin up an interactive container and (`-d`) to launch a daemon container. Then, we created a file, while inside our container, stopped it, and then started it back up to find our file still intact. Once we finished, we appropriately removed all our previous containers and images. Now, we are ready to move on to a new project, but before we do, and this will be important in every lab, do a general reset. Stop each container started during this lab. Ask your instructor if you need help doing this. Be proud of yourself!