**DESCRIPTION**

**Creating Containers**

**References**

* [docker create imagename](https://docs.docker.com/engine/reference/commandline/create/)
* [docker run imagename](https://docs.docker.com/engine/reference/commandline/create/)

**Introduction**

As we approach creating a container from our image, there are two paths we can take. In the instance where to set up the container requires a greater amount of configuration, it may be beneficial to create the container first and then move to the starting it up once prepared to do so. In most instances, however, we will build and run the container in one fell swoop with a single command. The former approach uses docker create and the latter uses docker run. Let's explore these ideas in more detail.

**> docker create imagename**

* can also be written as docker container create imagename
* creates a container that is in the "created" state - it configures and sets it up to be run including creating the *writable layer* on the image from which the container is created
* it's useful if there is some configuration of your container that you want to have ready, so you can easily just start up this new container
* it prints out the id of the newly created container
* flags:
  + -p hostport:containerport publish ports creates a mapping from the host port to the container ports
  + --env VARIABLE=value set environment variables of the container
  + -v hostdirectory:containerdirectory create a volume between the container and the host machine -- shared memory that persists past the given container's lifecycle
  + For example, let's pull an existing basic web app from Microsoft and create a container as described above.
* > docker docker pull mcr.microsoft.com/dotnet/core/samples:aspnetapp
* > docker create -p 80:80 mcr.microsoft.com/dotnet/core/samples:aspnetapp

Then typically you would then start the container.

> docker start firstthreecharactersinthecontainerid

If you follow the example above- navigate to localhost:80 on your web browser and see that your app has been started - with the container's port 80 being mapped to port 80 on your local machine.

**> docker run flags imagename**

* Can also be written docker container run flags imagename
* First pulls the image from the registry(repo of images) if it doesn't already exist locally.
* Then it both creates and runs the container-- this brings a container directly into the state of *running*.
* Recall that when you run a container its processes and environment are in their own isolated sandbox.
* To interact with a container's processes and work inside of its environment, you must tie the container to your own current environment- usually by having your commandline tied to a commandline within the container.
* Many of the flags influence the ways in which the container can be tied to your host environment and/or the way the initial container environment is set up, highlights include:
  + -d
    - Indicates to run the container in the background
    - If this flag is *omitted* the shell from which you are running the CLI commands will run in the *foreground*
      * That is, STDOUT and STDERR will be bound to the shell by default. Thus, anything that's printed to STDOUT/STDERR from within the container will display in the current shell. This does not mean that you can interact with Try to test the difference between running:
  + > docker run hello-world

And

> docker run -d hello-world

* + - The container's id prints out instead of the container's output.
    - In the second case even though the "Hello World..." output doesn't print, we can view the logs to see that it did in fact run and do what it was supposed to.

(viewing the logs is explained in [managing containers](https://app.revature.com/admin-v2/), but for now run the command with containerID replaced with the output from earlier.)

> docker container logs containerID

* + -i
    - allows you to run the container in interactive mode
    - it will keep the STDIN stream open even if your shell isn't attached to it.
    - it's needed if you want to be able to type input to your container dynamically
  + -t
    - is related to TTY and essentially is used to create some kind of terminal or shell
  + -it
    - This is the combination of the previous two commands and will be very frequently used when you want to open up a container with its primary processes being an interactive shell.
    - This allows you to basically work within the container from your commandline. For example try running:
  + > docker run -it ubuntu

Then you can try typing commands in your containerized ubuntu shell that you'll have open. You are essentially working inside of the container.

* + -a standardstream
    - this allows you to attach your commandline to a particular standard stream: STDIN, STDOUT, or STDERR
    - if you want to attach multiple then you will need multiple -a each followed by the particular stream
    - Also note, just attaching STDIN doesn't automatically make your container interactive. For that you must use the -i flag.
    - For example try testing the differences between attaching to just the error or just the standard output stream
  + > docker run -a STDERR -p 88:80 mcr.microsoft.com/dotnet/core/samples:aspnetapp

You should see your shell kind of paused- as though it's looking for some output, but not receiving any. Then stop your container- see [managing containers.](https://app.revature.com/admin-v2/)

>docker stop containerid

Then create and run a new container from the same image attaching a different stream:

>docker run -p 88:80 -a STDOUT mcr.microsoft.com/dotnet/core/samples:aspnetapp

Notice how now the standard output from your container displays in the shell.

* + -v name:directoryincontainer
    - Used to create a volume
    - To create a volume you name the volume and then you specify its corresponding path that will be linked within the container (the path must always begin with a / ).
    - Using this flag you can specify an absolute path on the host machine instead of a name (beginning with /), but that will create a *bind mount*.
    - volumes will persist after the container no longer exists
  + > docker run -it -v first\_volume:/example ubuntu

If inside of the newly created ubuntu shell, run

$ cd example

$ echo 'Hello World' > text\_file.txt

And then stop that container(CTL+C or CTL+D to exit the shell and thus stop the container's primary process exiting the container completely). Now spin up another one with the same volume

> docker run -it -v first\_volume:/data ubuntu

And then run the command

$ cat /data/text\_file.txt

It will print

Hello World

Notice/experiment with how this differs from if you created another ubuntu container from the ubuntu image without specifying the volume. It would not by default contain the folders. Those were only created and available because of the way that the volumes were configured.

* + -p portonhost:portincontainer
    - This command publishes a port in the container to a host port. This creates a tie between the two of them.
  + docker run -p 90:80 mcr.microsoft.com/dotnet/core/samples:aspnetapp

Then go to localhost:90 in the web browser to see site being hosted in the container.

* + --env VARIABLE=value
    - set the environment variables for the container
  + >docker run --env VAR1=value1 --env VAR2=value2 ubuntu env

Notice that the a whole list of environment variables is displayed, including the ones you just set.