Containerization Basics Presentation

Basic Containerization Demonstration

Docker help, info:

The purpose of this workshop is to familiarize you with the basics of the command line docker utilities, in particular image and container management. We'll find and pull images, run containers against those images, modify containers while showing statelessness of the images and define a new image.

(the final page is a list of resources and commands run in this demonstration)

To get the most basic information about what version of docker is running:

docker version

We can get a superset of '**docker version**' information along with configuration details, containers running, and so forth:

docker info

The '**--help**' option on any command or sub-command will give you typical linux style help documentation:

docker --help

We can run that on any command or sub-command to drill down on behavior and options as we will see.

Searching for Images:

Getting started identifying docker images you want to use is a bit like using a dictionary - you can't look a word up if you don't have some sense of how to spell it. Finding docker images can be done via their web hub, https://hub.docker.com/, which will give more context about the images - *if the publisher provides it!* We can do a bit of investigating from the command line but it is rudimentary.

Searching for images is done via the search sub-command:

docker search --help

Let's start with something classic and try to see what sort of hello world images we might have:

docker search hello

As you can see this returns a fair amount of results (25 is default). We can get the full description with '**--no-trunc**' and filter on 'official' images with a '**--filter**' argument:

docker search hello --no-trunc --filter "is-official=true"

This returns two results. We'll use the first one in our next steps.

Simple 'Hello-World' test:

Now that we have a hello-world image identified, let's run a simple test that bundles 'pulling' a docker image and starting a container based on it:

docker run hello-world

If this runs correctly it will have verified that docker is installed, configured and running correctly.

Note in the output that docker looks up and retrieves the image before running the container. Subsequent invocations would skip the 'pull' step. By executing this command we've found an image, pulled it into our local repository, and started a container based on it. So, let's look at what it's done, starting with images.

Image Management:

Image management is done via the **docker image** set of sub-commands, which we can list with '**--help**':

docker image --help

To see all images:

docker image ls -a

(alias **docker images**)

As noted previously, a call to **docker pull** was bundled into **docker run** when we ran the hello-world image. You can see that hello-world image in the local repository. We can investigate images through a few of the docker image sub-commands:

docker image history hello_world

The history will show us the layers in the image (the exact behavior is a bit different for images that you have created locally vs unmodified images pulled from a registry).

You can dig further into the details of an image with the inspect command, which will return a large structure of data that's beyond the scope of this demo:

docker image inspect hello_world

While we can let images be pulled implicitly when we execute docker run, we can also retrieve images independently. First let's remove the hello-world image:

docker image rm hello-world

Whoops! That triggers an error, because we still have a container that is using the hello-world. For now we'll remove that container so that we can continue with the images. Making note of the container name (note that docker assigns nonsensical names to the containers by default):

docker container ls -a

docker container rm <CONTAINER NAME>

docker image rm hello-world

Inspecting the local image registry, we can see that it's empty now:

docker image ls -a

Let's go ahead and pull the hello-world image again, explicitly:

docker pull hello-world

And now inspection of the local image registry will show the hello-world image again:

docker image ls -a

We can clean out images one at a time, but we also have the option to clear out any images in our current local registry in a single command:

docker image prune -a

This will prompt the user, but it can be forced to run without prompting with a '**-f**' argument

Container Management:

(We're going to need an image to work with, so let's "get" our hello-world image again):

docker run hello-world)

Image management is done via the **docker image** set of sub-commands, which we can list with '**--help**':

docker container --help

You can get more detailed help on the individual sub-commands by drilling down with the **'–-help'** option (as for images). In this case, let's look at the **'list'** sub-command

docker container list --help

This command will list all the containers active. It is aliased to **'docker ps'** and **'docker container ls'**. To force listing of all containers, including the idle ones (by default it just shows the active running containers), run with the **'-a'** option:

docker container ls -a

In this case we can see that we have a hello-world container still running, using the hello-world image.

As with images, you can get more information on a container with an inspect sub-command. Making note of the container name:

docker container inspect <CONTAINER NAME>

As for images, it returns a lot of structured data that is somewhat beyond the scope of this demo

Docker has assigned a random name to the container, we can rename it to something helpful to us:

docker container rename <OLD NAME> my_hello_world

docker ps -a

Our previous run of the hello-world container did not 'exit' when it's work (just printing a message to STDOUT) was done, but the instance is idle. Let's clean that up with the **rm** sub-command:

docker container rm my_hello_world

If we have multiple stopped or idle containers, we can get rid of them all in one go with a prune sub-command:

docker container prune

This will prompt you to be sure, but you can run it with a '**-f**' argument to force it without prompting, as for the image prune sub-command. If we take a look at our containers (I am going to use the shorter '**ps**' alias), we can see that there are none at present.

docker ps -a

Modifying a Container:

Let's put this together to do something a bit more interesting. We'll demonstrate the statelessness of the images, how to quickly generate a new image based on an existing one, and how to 'see' content within a container from outside. To do so we are going to run a simple webserver, **nginx**, in a dedicated container.

Work done within a container does not persist without deliberate efforts. Modified or new data files can be exposed (later demos will show this). But if the intent is to modify the state of the container and have it persist, then the thing to do is to define a new image.

We can do that by modifying a container and then committing it as a new image. The new image will be a derivative of the initial image the container was based on. First, retrieve our image:

docker pull nginx

This looked up the **nginx** image at the docker repository and pulled the latest version by default. We can start a **nginx** container as follows:

docker run -d -p 80:80 --name c_nginx nginx

This invocation has done a few things for us:

- **-name c_nginx**: This has given our nginx container a deliberate name, c_nginx.
- **-p 80:80**: This has mapped the ECS instances' external port 80 to the containers internal port 80. We can now navigate to the instances url and see a minimal webpage:
- **-d**: runs the container in the background

In a browser, navigate to <YOUR INSTANCE ADDRESS> and you should see this:

If you lose track of, or are curious if your container has certain port mappings, you can get a list of them as follows:

docker port c_nginx

Now, let's try to modify the container. To do so, we'll need to ssh to it. This is done by telling docker to execute a bash command in an interactive tty session. Having done our homework, we know that the webpage we want to edit is in /usr/share/nginx/html, so we can start there:

docker exec -it -w /usr/share/nginx/html c_nginx bash

The primary arguments to **docker exec** are the name of the container, **c_nginx**, and the command we want to execute - in this case the **bash** shell.

In addition we specified some additional arguments:

- **-w <some path>**: the -w argument tells us what the working directory for the command will be.
- **-i**: this will be an interactive session, it tells the container to keep STDIN open
- **-t**: sets up pseudo TTY session.

Now, from *within the container* let's modify the nginx webpage to say something a bit different:

Editing contents of a container:

```
cp index.html index.html.orig

cat index.html | sed 's/Welcome to nginx/Goodbye cruel world/g' > index.html.new

mv index.html.new index.html
```

If you go to your browser and shift-reload the web page, you will see that the title has changed per our modifications.

We are now done with our work within the container so we will exit.

Exit the container

```
exit
```

We're going to shut that container down to show that the state is lost, but first let's pull the modified file out and retain it.

We can see changes to a docker container by running a **diff** sub-command:

docker container diff c_nginx

Alias: **docker diff c_nginx**

This prints out a list of added (A), changed (C) or deleted (D) files (of which we have none) relative to the image that the container is based off of. We can see a number of changes and additions, most of which are not related to the change we made, but have to do with activities the container took when it was started. But we do see our edits listed:

- /usr/share/nginx/html : every directory in this path shows a change because there were changes to files within that directory
- /usr/share/nginx/html/index.html : has category C, as it is now different
- /usr/share/nginx/html/index.html.orig : has category A, as we added it to the container.

We can extract the modified file using a '**cp**' sub-command:

docker container cp c_nginx:/usr/share/nginx/html/index.html index.html

The key argument here is 'c_nginx:/usr/share/nginx/html/index.html'. This addresses the target container (c_nginx) and then the full file path inside it.

Let's stop the container:

docker container stop c_nginx

docker container rm c_nginx

If we attempt to navigate to the url, we will see that the server has been stopped. If we run our docker container query, we see that non are active:

docker ps -a

Define an Image:

Let's take the file we extracted from the **c_nginx** container and use it as the basis for a new image.

Let's start a new **nginx** container, one that will be a source for our new **sad_nginx** image.

docker run -d -p 80:80 --name c_sad_nginx nginx

Once again, let's navigate to port 80 of our instance. Having started a fresh container off the same original image, our edits are gone.

Let's inject our modified web page into the fresh **nginx** container we've started

docker cp index.html c_sad_nginx:/usr/share/nginx/html/index.html

In this case we have used the '**cp**' sub command not to extract the modified file from the container, but to inject it into the container.

Reloading the web page, we now see our edits restored without having had to go in and make them manually.

Rather than having to perform the cp command every time, we can preserve our edits by making a new image based off of a container. This is done, pretty simply, through the '**commit**' sub-command:

docker container commit c_sad_nginx i_sad_nginx:latest

In this case we have created a new image, **i_sad_nginx**, based off the **c_sad_nginx** container, and attached a **latest** tag to it. We can see it in the list of images in our local registry:

docker images

Let's prove to ourselves that the new image retains the modifications. We'll stop the **c_sad_nginx** container, which is a modified instance of the original **nginx** image, and start a new container based on our local **i_sad_nginx** image. We can get around running sequential '**stop**' and '**rm**' sub-commands by issuing the '**rm**' sub-command with a '**-f**' argument that will force the container's removal (effectively, it bundles a '**container kill**' with the '**rm**' command):

docker container rm -f c_sad_nginx

docker run -d -p 80:80 --name c_sad_nginx i_sad_nginx

Navigating to the port again, we see our modified webpage based on our new, local image.

Finally, we can save an image outside the image registry and later re-import it.

docker image save i_sad_nginx > sad_nginx.tar

If we clear out our image registry, we can then re-import the tar file and build a container against it. We'll remove the **c_sad_nginx** container again.

docker container rm -f c_sad_nginx

Then we clean our our image registry:

docker image prune -a

docker images -a

Let's import our tarfile and see if we can start a container against it:

docker image load -i sad_nginx.tar

The '**-i**' argument forces it to read from a tar file instead of STDIN. If we look at our images (I am going to use the '**images**' alias which is slightly shorter), we can see it has been imported.

docker images -a

Re-running our container command, we can navigate to our instance and see the modified web server running:

docker run -d -p 80:80 --name c_sad_nginx i_sad_nginx

(return to the browser and reload to see our sad greeting)

Resources, commands

The following are some of the resources I used in putting together this presentation:

https://hackernoon.com/running-docker-on-aws-ec2-83a14b780c56

https://aws.amazon.com/getting-started/tutorials/deploy-docker-containers/

https://docs.docker.com/glossary/

https://docs.docker.com/get-started/

The following is a complete list of the commands executed in this demo:

Help/Info:
docker version
docker info
docker --help

Searching:
docker search --help
docker search hello
docker search hello --no-trunc --filter "is-official=true"

Simple Test:
docker run hello-world

Image Management:
docker image --help
docker image ls -a
docker images
docker image history hello-world
docker image inspect hello-world
docker image rm hello-world
docker ps -a
docker container rm <CONTAINER NAME>
docker image rm hello-world
docker image ls -a
docker pull hello-world
docker image ls -a
docker image prune -a

Container Management:
docker run hello-world
docker container --help
docker container list --help
docker container ls -a
docker container inspect relaxed_noether
docker container rename relaxed_noether my_hello_world
docker ps -a
docker container rm my_hello_world
docker container prune --help
docker container prune
docker ps -a

Container Modification:
docker pull nginx
docker container run -d -p 80:80 -name c_nginx nginx
docker container run -d -p 80:80 --name c_nginx nginx
docker port c_nginx
docker exec -it -w /usr/share/nginx/html c_nginx bash
docker container diff c_nginx
docker diff c_nginx
docker container cp c_nginx:/usr/share/nginx/html/index.html index.html
docker container stop c_nginx
docker container rm c_nginx
docker ps -a

Image Definition:
docker run -d -p 80:80 --name c_sad_nginx nginx
docker cp index.html c_sad_nginx:/usr/share/nginx/html/index.html
docker container commit c_sad_nginx i_sad_nginx:latest
docker images
docker container rm -f c_sad_nginx
docker run -d -p 80:80 --name c_sad_nginx i_sad_nginx
docker image save i_sad_nginx > sad_nginx.tar
docker container rm -f c_sad_nginx
docker image prune -a
docker images
docker image ls -a
docker image load -i sad_nginx.tar
docker images -a
docker run -d -p 80:80 --name c_sad_nginx i_sad_nginx
docker ps -a

Cleanup:
docker container rm -f c_sad_nginx
docker ps -a
docker image prune -a
docker images -a