Introduction to Docker

Running containers

Let's first take a look at running a container. Open your terminal and run the following command.

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

As you can read in the message that is printed in your terminal, Docker did the following:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

Now let's take a look at running a basic web server. We'll use the Official NGINX image hosted on Docker Hub. Run the following command to start the web server on your computer.

```
$ docker run --detach --publish 8080:80 --name web nginx
```

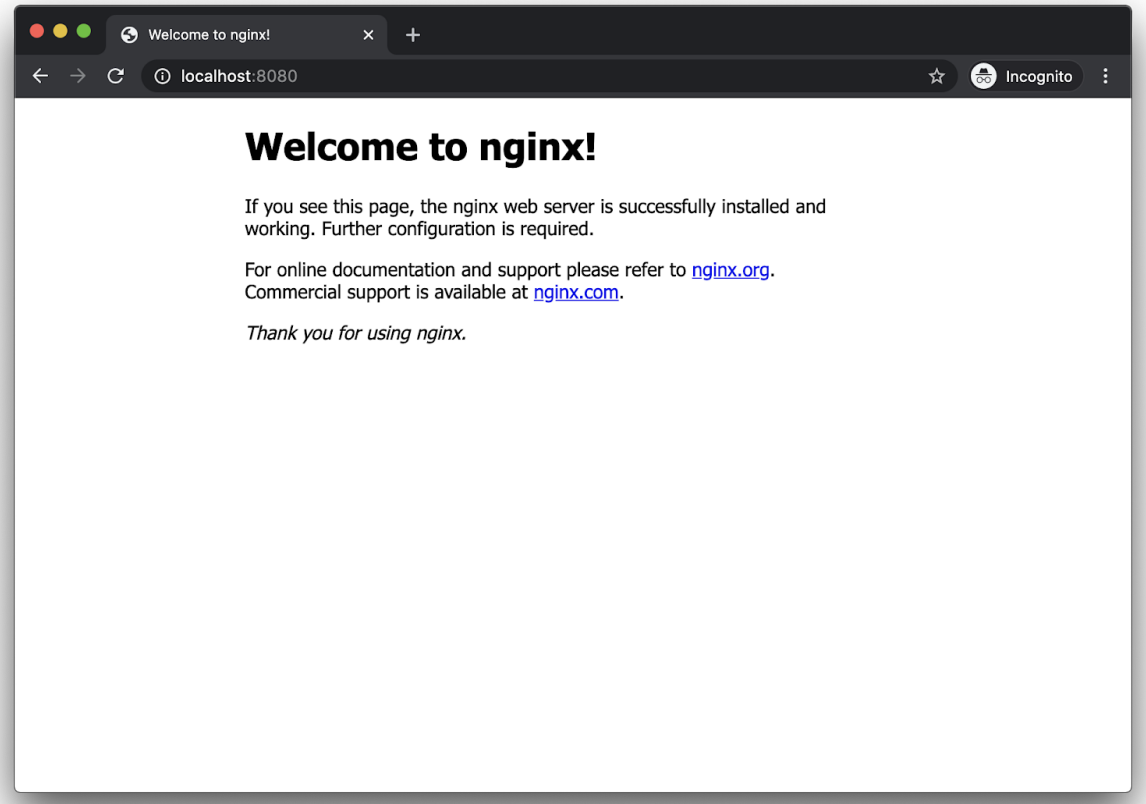
The above command uses the `--detach` flag which tells Docker to detach from the process and run it in the background.

As we mentioned in the introduction, images are run inside of a "container". A container is an isolated process which includes the isolation of networking. Therefore to allow traffic from outside our container, we use the `--publish` flag. The `--publish` flag tells Docker to expose or publish a container port to the local host network. So in the command above, we are connecting port 8080 on the local network to port 80 inside our container.

When Docker starts a container, it will give it a random name. To allow us to more easily manage our containers, we explicitly give it a name using the `--name` flag.

Open your favorite web browser and navigate to http://localhost:8080. You will see the default NGINX welcome page.

> ***NOTE**: If you are working on a remote VM and do not have a web browser, you can use the following curl command to test that your container is running properly.*
>
> *$ curl http://localhost:8080*



Stopping, starting and removing containers

At this point, we've taken a look at running an image inside of a container. Now let's take a look at managing our containers.

To get a list of running containers on the system, run the following command.

```
$ docker ps
```

Docker prints the running containers on your system and displays the following columns:

.

- CONTAINER ID - This is the unique identifier of your container.
- IMAGE - Displays the image that is running inside the container.
- COMMAND - Displays the command that was used to start the process running inside your container.
- CREATED - When the container was started.
- STATUS - The status of your container. Typically how long your container has been running or how long it has been stopped.

- PORTS - Displays the ports that have been published.
- NAMES - The unique container name.

If we want to see all containers on your system whether they are running or not, we can pass the `--all` flag or the short hand `-f`.

```
$ docker ps -a
```

You should now see two containers listed. One for th NGINX web server and the other for the hello-world image we ran earlier.

As you can see, the hello-world image has exited. Containers can be in multiple states. The most frequently used states are “Up” or “Exited”. When a container is stopped, it is not removed and therefore will still be listed in the list of containers.

Let’s start the hello-world container. Find the CONTAINER ID of the hello-world container and subtituite it in the command below.

```
$ docker start -i <CONTAINER ID>
```

The container will start back up and display the same message that we saw earlier.

Let’s stop the container that is running our NGINX server. Find the name of the NGINX container that we ran earlier and substitute it in the command below.

```
$ docker stop <NAME>
```

Let’s take a look at the containers and their status.

```
$ docker ps -a
```

Both of our containers are now in an “Exited” status. Let’s totally remove them. Find the CONTAINER ID of the hello-world container and find the container NAME of the NGINX container. You can use either of those identifiers with the rm (remove) command below. Run the following command for both of your containers to remove them.

```
$ docker rm <CONTAINER ID/NAME>
```

List your container again and confirm they have been removed.

```
$ docker ps -a
```

Managing images

Just like we are able to list out the containers on our system, we can also list the images that are located on our system.

```
$ docker images
```

Docker prints the following details about the images located on your system:

- REPOSITORY - The full name of the image.
- TAG - The specific tag of the image.
- IMAGE ID - Unique identifier of the image.
- CREATED - When the image was created.
- SIZE - General size of the image.

We can also tag an image with a different name and tag. Let's give the hello-world image a different name and tag. To do that, we'll use the tag command which takes the image you want to use to create the new tag with and then the name and tag of the new image.

```
$ docker tag hello-world my-hello
```

Now list your images again.

```
$ docker images
```

Now let's take a look at sharing this image using Docker Hub. Login into Hub using your Docker ID and password you created in the prerequisites.

```
$ docker login --username
```

Before we can push an image to Hub, we need to tag it with your Docker ID. Let's do that now and then push to the Hub registry.

```
$ docker tag my-hello <docker id>/my-hello
$ docker push <docker id>/my-hello
```

Open http://hub.docker.com in your browser and login. You will see the my-hello image that you just pushed.

Now that our image is shared on Hub, we can share it with our coworkers or the public. Let's delete the image we have now and pull the image from Hub.

```
$ docker rmi <docker id>/my-hello
```

The rmi (remove image) command will delete the image from your system. To check our work, well list the images we have locally.

```
$ docker images
```

Now let's pull the image from Docker Hub.

```
$ docker pull <docker id>/my-hello
```

Let's remove all the images on our system.

```
$ docker rmi -f $(docker images -q -a)
```

In the above command, we used the `rmi` command from above passing in the result of the docker images command. To only return the IMAGE ID of a container, we pass the -q and -a flags to the docker images command.

Creating images

We've taken a look at how to run and manage containers. We also took a look at managing and sharing images. Now let's take a look at creating our own image.

A `Dockerfile` is a text document that contains all the commands a user could call on the command line to assemble an image. When we tell Docker to build our image by executing the `docker build` command, Docker will read these instructions and execute them one by one and create a Docker image as a result.

Let's walk through creating a `Dockerfile` for our application. In the root of your working directory, create a file named `Dockerfile` and open this file in your text editor.

> ***NOTE:*** *The name of the Dockerfile is not important but the default filename for many commands is simply* `Dockerfile`*. So we'll use that as our filename throughout this series.*

The first thing we need to do is add a line in our Dockerfile that tells Docker what base image we would like to use for our application.

Dockerfile:

```
FROM node:12.18.1
```

Docker images can be inherited from other images. So instead of creating our own base image, we'll use the official Node.js image that already has all the tools and packages that we need to run a Node.js application. You can think of this as in the same way you would think about class

inheritance in object oriented programming. So for example. If we were able to create Docker images in JavaScript, we might write something like the following.

```
class MyImage extends NodeBaseImage {}
```

This would create a class called `MyImage` that inherited functionality from the base class `NodeBaseImage`.

In the same way, when we use the `FROM` command, we tell docker to include in our image all the functionality from the `node:12.18.1` image.

NOTE: If you want to learn more about creating your own base images, please checkout our documentation on creating base images.

To make things easier when running the rest of our commands, let's create a working directory. This instructs Docker to use this path as the default location for all subsequent commands. This way we do not have to type out full file paths but can use relative paths based on the working directory.

```
WORKDIR /app
```

Usually the very first thing you do once you've downloaded a project written in Node.js is to install npm packages. This will ensure that your application has all its dependencies installed into the `node_modules` directory where the node runtime will be able to find them.

Before we can run npm install, we need to get our `package.json` and `package-lock.json` files into our images. We'll use the `COPY` command to do this. The `COPY` command takes two parameters. The first parameter tells Docker what file(s) you would like to copy into the image. The second parameter tells Docker where you want that file(s) to be copied to. We'll copy the `package.json` and `package-lock`.json file into our working directory - /app.

```
COPY package.json package.json
COPY package-lock.json package-lock.json
```

Once we have our package.json files inside the image, we can use the `RUN` command to execute the command `npm install`. This works exactly the same as if we were running npm install locally on our machine but this time these node modules will be installed into the node_modules directory inside our image.

```
RUN npm install
```

At this point we have an image that is based on node version 12.18.1 and we have installed our dependencies. The next thing we need to do is to add our source code into the image. We'll use the COPY command just like we did with our package.json files above.

```
COPY . .
```

This COPY command will take all the files located in the current directory and copies them into the image. Now all we have to do is to tell Docker what command we want to run when our image is run inside of a container. We do this with the CMD command.

```
CMD [ "node", "server.js" ]
```

Below is the complete Dockerfile.

```
FROM node:12.18.1

WORKDIR /app

COPY package.json package.json
COPY package-lock.json package-lock.json

RUN npm install

COPY . .

CMD [ "node", "server.js" ]
```

Building Images

Now that we've created our Dockerfile, let's build our image. To do this we use the docker build command. The `docker build` command builds Docker images from a Dockerfile and a "context". A build's context is the set of files located in the specified PATH or URL. The Docker build process can access any of the files located in the context.

The build command optionally takes a --tag flag. The tag is used to set the name of the image and an optional tag in the format 'name:tag'. We'll leave off the optional "tag" for now to help simplify things. If you do not pass a tag, docker will use "latest" as it's default tag. You'll see this in the last line of the build output.

Let's build our first Docker image.

```
$ docker build --tag node-docker .

Sending build context to Docker daemon  82.94kB
```

```
Step 1/7 : FROM node:12.18.1
---> f5be1883c8e0
Step 2/7 : WORKDIR /code
...
Successfully built e03018e56163
Successfully tagged node-docker:latest
```

To see the image we just create, simply run the `images` command.

```
$ docker images
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
node-docker         latest              3809733582bc        About a
minute ago   945MB
node                12.18.1             f5be1883c8e0        2 months
ago         918MB
```

You should see at least two images listed. One for the base image `node:12.18.1` and the other for our image we just build `node-docker:latest`.

Tagging Images

As mentioned earlier, an image name is made up of slash-separated name components. Name components may contain lowercase letters, digits and separators. A separator is defined as a period, one or two underscores, or one or more dashes. A name component may not start or end with a separator.

An image is made up of a manifest and a list of layers. Do not worry to much about manifests and layers at this point other than a “tag” points to a combination of these artifacts. You can have multiple tags for an image. Let’s create a second tag for the image we built and take a look at it’s layers.

To create a new tag for the image we built above, run the following command.

```
$ docker tag node-docker:latest node-docker:v1.0.0
```

The docker tag command creates a new tag for an image. It does not create a new image. The tag points to the same image and is just another way to reference the image.

Now run the `docker images` command to see a list of our local images.

```
$ docker images
```

```
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
node-docker         latest              3809733582bc        24
minutes ago      945MB
node-docker         v1.0.0              3809733582bc        24
minutes ago      945MB
node                12.18.1             f5be1883c8e0        2 months
ago        918MB
```

You can see that we have two images that start with `node-docker`. We know they are the same image because if you look at the `IMAGE ID` column, you can see that the values are the same for the two images.

Let's remove the tag that we just created. To do this, we'll use the `rmi` command. The `rmi` command stands for "remove image".

```
$ docker rmi node-docker:v1.0.0
Untagged: node-docker:v1.0.0
```

Notice that the response from Docker tells us that the image has not been removed but only "untagged". Double check this by running the images command.

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
node-docker         latest              3809733582bc        32
minutes ago      945MB
node                12.18.1             f5be1883c8e0        2 months
ago        918MB
```

Our image that was tagged with `:v1.0.0` has been removed but we still have the `node-docker:latest` tag available on our machine.

Conclusion

In this module, we took a look at running and managing containers, creating and managing images and building images using Dockerfiles. Then we took a look at tagging our images and removing images.