CS312 :: Lab Week 5 :: WordPress on Docker

In this lab, we'll install and configure a couple Docker containers, link them together, and start them running.

Supplies needed

- Personal laptop with VirtualBox installed
- The CentOS7 DVD .ISO available on our Canvas page

Perform the Tasks and Problems in order, as given below.

Procedure

WordPress is a very popular blogging platform used by millions of writers, reporters, and editors around the world. It is also a relatively simple app to deploy via containers. But before we can do that, we need a Linux VM with a GUI to make this all easier.

Create Virtual Machine

On your laptop, create a normal VirtualBox CentOS 7 VM, following the instructions as laid out in the lectures. Do this quick: it will take about 10 to 15 minutes for this VM creation to finish. There is plenty of time, however, to complete this Lab.

As you create the CentOS7 VM, this time, however, when you get the Installation Summary page, and you're able to select the Network, Time, etc., click on the Software Selection button. In this area, click GNOME Desktop on the left-hand side, then click Done. Click Begin Installation, and your CentOS VM will be created with a GUI! We recommend that you set the root password as "password" and set up a user with username "centosuser" and password "password", all for simple testing purposes.

Make sure that you select the option that creates your user as an Administrator. If you forget, and the VM completes installation, you won't be able to use sudo. To fix this, follow this procedure in the Terminal app in the VM:

1. `$ su`
   (Log in as root)
2. `usermod -aG wheel centosuser`
3. `exit`
4. Log out and log back in

Once you're logged in, quickly follow this procedure to get Docker installed and configured, again typing it into the Terminal:

1. From a terminal, run: `$ sudo yum install -y yum-utils device-mapper-persistent-data lvm2`
2. `$ sudo yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo`
3. `$ sudo yum install -y docker-ce`
4. `$ sudo systemctl start docker`
5. `$ sudo systemctl enable docker`
6. `$ sudo usermod -aG docker centosuser`
7. Log out and back on
8. Test that docker is working by running: `$ docker run hello-world`

WordPress Intro

We're going to install WordPress in three different ways:

1. Docker CLI with Linking
2. Docker CLI with Networks
3. Docker Stack

The purpose of this is to expose you to some of the container-based networking capabilities Docker has and demonstrate some of the rudimentary orchestration tools Docker has out of the box. All of the commands that follow use the Terminal app, and a few use Firefox, which came installed by default in your VM's GUI.

Docker Linking

1. WordPress requires MySQL to store data. Instead of running MySQL natively on the host OS, we're going to run it in a separate container, and **link** this MySQL container to a WordPress container.

   Starting a MySQL container with Docker is relatively simple. In your CentOS VM, run the following command to start version 5.7.22:

   ```
   $ docker run --name wordpress-mysql --rm
   -e MYSQL_ROOT_PASSWORD=mypasswd -d mysql:5.7.22
   ```

   These switches and parameters should be known to you from the lectures earlier this week, although the -e switch might be new. This one sets up an environment variable called "MYSQL_ROOT_PASSWORD" and sets it to "mypasswd". In this case, this environment variable is automatically made accessible to any other container that "links".

2. Starting a WordPress container is also pretty easy. We're going to "link" our WordPress container with the existing MySQL container. This exposes environments between the containers and gives them easy networking. We'll also expose the WordPress container's port 80 to our controlling CLI host as port 8080.

   Fire up the WordPress version 4 container like this:

   ```
   $ docker run --name wordpress-main --rm -p 8080:80 --link
   wordpress-mysql:mysql -d wordpress:4
   ```

   This WordPress install *needs* the root mysql server password, and it knows the environment variable it should be stored in - and because of the link, it's able to access that data automatically!

   To test this out, open up Firefox in the CentOS VM's GUI, go to "localhost:8080", and see the WordPress set up procedure's first page which asks for a language choice.

   QUESTION: Go answer Question 1 now.

3. Go ahead and shut down the containers. To cleanup your running containers run the following commands:

   ```
   $ docker stop wordpress-main
   $ docker stop wordpress-mysql
   ```

Docker Networks

1. Docker **networks** can be thought of as a private network that containers can be added into. This is different from **linking** in that you can assign custom hostnames, IP addresses, etc. to different containers. Linking also shares additional data between containers, like environment variables, which might not always be desired.

   Here we'll try pinging the container "some-service" in the network called "some-project". Type in these three lines:

   ```
   $ docker network create some-project
   $ docker run --rm --name some-service --network some-project -d
   alpine /bin/sleep 10000
   $ docker run alpine ping -c 3 some-service
   ping: bad address 'some-service'
   ```

   The first line creates a network that will automatically assign IP addresses to joining containers. We call this network simply "some-project".

   The next line creates an alpine container named "some-service", which corresponds to the NETBIOS name of the computer. Think of this as the name you'd assign the computer, like "MIKESLAPTOP" or "FAMILYPC" or "MYSQLSERVER". Typically, computers are ping-able by this name, like "`ping os1`". You can also see that this container is put onto the "some-project" network by the "`--network`" switch.

   Take a look at the third command line: what happens as a result of running this is that even though the sleeping alpine container is put onto the "some-project" network, the *second* alpine container that is trying to ping the first is not in the same network, because the second one's command didn't specify as such. Thus, the ping command fails. We can verify that only one computer is connected to the "some-project" network, because we can inspect the network to see who's connected.

   Run this command:
   ```
   $ docker network inspect some-project | more
   ```

   QUESTION: Go answer Question 2 now.

   Now, using the information above, you need to experiment and create a single line that runs an alpine container in the "some-project" network that successfully pings the "some-service" container 3 times. Test it out!

   QUESTION: Go answer Question 3 now.

   Go ahead and shut down your service container, and remove the network:

   ```
   $ docker stop some-service
   $ docker network rm some-project
   ```

2. Now that we've seen how Docker networks work, let's create a network for the WordPress containers:

   ```
   $ docker network create wpnet
   ```

Easy! You can verify that this is up and running with:

```
$ docker network ls
```

Next, we'll start up our MySQL container again. Note that we add the "`--network`" flag to set the network this container will work on.

```
$ docker run --rm --name wordpress-mysql -e MYSQL_ROOT_PASSWORD=
mypasswd --network wpnet -d mysql:5.7.22
```

Let's now start the WordPress container. Note again that we are setting the network *and* setting some environment variables this time, which is different from when we just linked the containers. These were implicitly shared when the containers were linked. Since we're now using networks, these values need to be set explicitly.

```
$ docker run --rm --name wordpress-main -p 8080:80 -e
WORDPRESS_DB_HOST=wordpress-mysql:3306 -e
WORDPRESS_DB_PASSWORD=mypasswd --network wpnet -d wordpress:4
```

Again, go to localhost:8080 in the browser of your VM (note that `curl` doesn't show this WP start page for some reason) and verify that everything is working. Get the next question answered before you close the browser.

QUESTION: Go answer Question 4 now.

To cleanup your running containers run the following commands:

```
$ docker stop wordpress-main
$ docker stop wordpress-mysql
$ docker network rm wpnet
```

Automation with Docker Stack

1. Docker Stack is a tool for deploying related containers together. It's also an easy way to avoid saving long "`docker run ...`" shell commands because you store everything in YAML configuration files.

   To get started, first enable Docker **Stack** by starting a local Docker **Swarm**. Yes, this is all pretty buzz-word-ey. Welcome to modern tech.

   Create the Docker Swarm by running the following command:

   ```
   $ docker swarm init
   ```

   This creates a local Swarm, with this docker installation on this VM as the Swarm **Manager**. This local swarm is in contrast to a cloud-provided Docker Swarm, which would be used to easily deploy your containerized application to a server provided by Amazon Web Services, Google Compute Engine, or DigitalOcean, all collectively called Platform as a Service (PaaS).
   Unfortunately using PaaS is outside the scope of this course; but it is pretty cool stuff (if not ridiculously complicated and platform-specific).

2. Next, copy the following text into a file called “stack.yaml” on the same VM you've been working on. Pay attention to the double spaces that are used as the indent. We recommend you store this in “~/stack.yaml”:

```
version: '3'
services:
  wordpress:
    image: wordpress:4
    ports:
      - '8080:80'
    networks:
      - wpnet
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure
    environment:
      WORDPRESS_DB_PASSWORD: mypasswd

  mysql:
    image: mysql:5.7.22
    networks:
      -wpnet
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure
    environment:
      MYSQL_ROOT_PASSWORD: mypasswd

networks:
  wpnet:
```

Hint: this file is also available at:
http://web.engr.oregonstate.edu/~brewsteb/CS312/stack.yaml

This file does a lot of things:

- Creates two services, “wordpress” and “mysql” (note how these two elements are nested under the “services” element)
- Exposes the “wordpress” port to the host (our VM)
- Deploys 1 replica of the “wordpress” frontend
- Tells Docker when to restart the container
- Sets the environment variable “WORDPRESS_DB_PASSWORD”
- Does all that same stuff for the MySQL container
- Creates the “wpnet” network
- Adds all containers that refer to it to the “wpnet” network

3. Run the following commands to start this **Stack** up. Here, we'll start the WordPress and MySQL services under the `wp` namespace:

```
$ docker stack deploy -c stack.yaml wp
$ docker stack ls                # Look at your deployed services
$ docker stack ps wp             # Inspect a specific service
$ docker service inspect wp_mysql # Get a in-depth review of wp_mysql
```

Note: don't worry if you can't see this stack from your web browser - there may be some version conflicts that prevent it from showing. This doesn't affect what we're doing in the rest of the lab.

QUESTION: Go answer Question 5 now.

As our last task, let's try to kill the WordPress container (not the MySQL one), and observe the Docker service that's watching over both containers bring it back up. To do this, first run:

```
$ docker ps
```

This will return a list of the containers running which will include both the wordpress and mysql containers. Note that there is a 12-character, hexadecimal Container ID on the left side. Find the Container ID for the WordPress container and run a Kill command against it, like so:

```
$ docker kill <12-char cont. ID>
```

Now, start running "`docker ps`" about every second. You'll see that the first few entries return just the MySQL container. However, after around 5-10 seconds, you should see a result that lists BOTH containers running again.

QUESTION: Go answer Question 6 now.

To cleanup, run:

```
$ docker stack rm wp
```

Questions

1) Get the TAs initials, showing you got to the WordPress setup page. *(7 points)*
2) What IPv4 address does the "some-service" container have? *(6 points)*
3) What single docker command starts up a container that successfully pings the "some-service" container with 3 ping packets, then removes itself after it runs? *(8 points)*
4) Get the TAs initials, showing you got to the WordPress setup page via the Docker network. *(6 points)*
5) Get the TAs initials, showing that the "`docker stack ps wp`" command works (*6 points*)
6) Get the TAs initials, showing that the "`docker ps`" command shows the two containers having a different pp time in the Status field. *(7 points)*

You're done! To receive credit for this lab, you must turn in your answer sheet.