Docker Orchestration Hands-on Lab

In this lab you will play around with the container orchestration features of Docker. You will deploy a simple application to a single host and learn how that works. Then, you will configure Docker Swarm Mode, and learn to deploy the same simple application across multiple hosts. You will then see how to scale the application and move the workload across different hosts easily.

Difficulty: Beginner

Time: Approximately 30 minutes

Tasks:

Section 1: What is Orchestration

So, what is Orchestration anyways? Well, Orchestration is probably best described using an example. Let's say that you have an application that has high traffic along with high-availability requirements. Due to these requirements, you typically want to deploy across at least 3+ machines, so that in the event a host fails, your application will still be accessible from at least two others. Obviously, this is just an example and your use-case will likely have its own requirements, but you get the idea.

Deploying your application without Orchestration is typically very time consuming and error prone, because you would have to manually SSH into each machine, start up your application, and then continually keep tabs on things to make sure it is running as you expect.

But, with Orchestration tooling, you can typically off-load much of this manual work and let automation do the heavy lifting. One cool feature of Orchestration with Docker Swarm, is that you can deploy an application across many hosts with only a single command (once Swarm mode is enabled). Plus, if one of the supporting nodes dies in your Docker Swarm, other nodes will automatically pick up load, and your application will continue to hum along as usual.

If you are typically only using `docker run` to deploy your applications, then you could likely really benefit from using Docker Compose, Docker Swarm mode, or both Docker Compose and Swarm.

Section 2: Configure Swarm Mode

Real-world applications are typically deployed across multiple hosts as discussed earlier. This improves application performance and availability, as well as allowing individual application components to scale independently. Docker has powerful native tools to help you do this.

An example of running things manually and on a single host would be to create a new container on **node1** by running `docker run -dt ubuntu sleep infinity`.

```
docker run -dt ubuntu sleep infinity
```

```
Unable to find image 'ubuntu:latest' locally

latest: Pulling from library/ubuntu

d54efb8db41d: Pull complete

f8b845f45a87: Pull complete

e8db7bf7c39f: Pull complete

9654c40e9079: Pull complete

6d9ef359eaaa: Pull complete

Digest: sha256:dd7808d8792c9841d0b460122f1acf0a2dd1f56404f8d1e56298048885e45535

Status: Downloaded newer image for ubuntu:latest

846af8479944d406843c90a39cba68373c619d1feaa932719260a5f5afddbf71
```

This command will create a new container based on the `ubuntu:latest`image and will run the `sleep` command to keep the container running in the background. You can verify our example container is up by running `docker ps` on **node1**.

```
docker ps

CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES

044bea1c2277        ubuntu              "sleep infinity"    2 seconds ago       Up
1 second                            distracted_mayer
```

But, this is only on a single node. What happens if this node goes down? Well, our application just dies and it is never restarted. To restore service, we would have to manually log into this machine, and start tweaking things to get it back up and running. So, it would be helpful if we had some type of system that would allow us to run this “sleep” application/service across many machines.

In this section you will configure *Swarm Mode*. This is a new optional mode in which multiple Docker hosts form a self-orchestrating group of engines called a *swarm*. Swarm mode enables new features such as *services* and *bundles* that help you deploy and manage multi-container apps across multiple Docker hosts.

You will complete the following:

- Configure *Swarm mode*
- Run the app
- Scale the app
- Drain nodes for maintenance and reschedule containers

For the remainder of this lab we will refer to *Docker native clustering* as ***Swarm mode***. The collection of Docker engines configured for Swarm mode will be referred to as the *swarm*.

A swarm comprises one or more *Manager Nodes* and one or more *Worker Nodes*. The manager nodes maintain the state of swarm and schedule application containers. The worker nodes run the application containers. As of Docker 1.12, no external backend, or 3rd party components, are required for a fully functioning swarm - everything is built-in!

In this part of the demo you will use all three of the nodes in your lab. **node1** will be the Swarm manager, while **node2** and **node3** will be worker nodes. Swarm mode supports highly available redundant manager nodes, but for the purposes of this lab you will only deploy a single manager node.

Step 2.1 - Create a Manager node

In this step you'll initialize a new Swarm, join a single worker node, and verify the operations worked.

Run `docker swarm init` on **node1**.

```
docker swarm init --advertise-addr $(hostname -i)
```

```
Swarm initialized: current node (6dlewb50pj2y66q4zi3egnwbi) is now a manager.


To add a worker to this swarm, run the following command:


    docker swarm join \

    --token SWMTKN-1-1wxyoueqgpcrc4xk2t3ec7n1poy75g4kowmwz64p7ulqx611ih-
68pazn0mj8p4p4lnuf4ctp8xy \

    10.0.0.5:2377


To add a manager to this swarm, run 'docker swarm join-token manager' and follow
the instructions.
```

You can run the `docker info` command to verify that **node1** was successfully configured as a swarm manager node.

```
docker info
```

```
Containers: 2
 Running: 0
 Paused: 0
 Stopped: 2
Images: 2
Server Version: 17.03.1-ee-3
Storage Driver: aufs
 Root Dir: /var/lib/docker/aufs
 Backing Filesystem: extfs
 Dirs: 13
 Dirperm1 Supported: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
 Volume: local
 Network: bridge host macvlan null overlay
Swarm: active
```

```
 NodeID: rwezvezez3bg1kqg0y0f4ju22
 Is Manager: true
 ClusterID: qccn5eanox0uctyj6xtfvesy2
 Managers: 1
 Nodes: 1
 Orchestration:
  Task History Retention Limit: 5
 Raft:
  Snapshot Interval: 10000
  Number of Old Snapshots to Retain: 0
  Heartbeat Tick: 1
  Election Tick: 3
 Dispatcher:
  Heartbeat Period: 5 seconds
 CA Configuration:
  Expiry Duration: 3 months
 Node Address: 10.0.0.5
 Manager Addresses:
  10.0.0.5:2377
<Snip>
```

The swarm is now initialized with **node1** as the only Manager node. In the next section you will add **node2** and **node3** as *Worker nodes*.

Step 2.2 - Join Worker nodes to the Swarm

You will perform the following procedure on **node2** and **node3**. Towards the end of the procedure you will switch back to **node1**.

Now, take the entire `docker swarm join ...` command we copied earlier from `node1` where it was displayed as terminal output. We need to paste the copied command into the terminal of **node2** and **node3**.

It should look something like this for **node2**. By the way, if the `docker swarm join ...` command scrolled off your screen already, you can run the `docker swarm join-token worker` command on the Manager node to get it again.

Remember, the tokens displayed here are not the actual tokens you will use. Copy the command from the output on **node1**. On **node2** and **node3** it should look like this:

```
docker swarm join \
    --token SWMTKN-1-1wxyoueqgpcrc4xk2t3ec7n1poy75g4kowmwz64p7ulqx611ih-
68pazn0mj8p4p4lnuf4ctp8xy \
```

```
    10.0.0.5:2377

docker swarm join \

    --token SWMTKN-1-1wxyoueqgpcrc4xk2t3ec7n1poy75g4kowmwz64p7ulqx611ih-
68pazn0mj8p4p4lnuf4ctp8xy \

    10.0.0.5:2377
```

Once you have run this on **node2** and **node3**, switch back to **node1**, and run a `docker node ls` to verify that both nodes are part of the Swarm. You should see three nodes, **node1** as the Manager node and **node2** and **node3** both as Worker nodes.

```
docker node ls
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
6dlewb50pj2y66q4zi3egnwbi *  node1   Ready   Active        Leader
ym6sdzrcm08s6ohqmjx9mk3dv    node3   Ready   Active
yu3hbegvwsdpy9esh9t2lr431    node2   Ready   Active
```

The `docker node ls` command shows you all of the nodes that are in the swarm as well as their roles in the swarm. The `*` identifies the node that you are issuing the command from.

Congratulations! You have configured a swarm with one manager node and two worker nodes.

Section 3: Deploy applications across multiple hosts

Now that you have a swarm up and running, it is time to deploy our really simple sleep application.

You will perform the following procedure from **node1**.

Step 3.1 - Deploy the application components as Docker services

Our `sleep` application is becoming very popular on the internet (due to hitting Reddit and HN). People just love it. So, you are going to have to scale your application to meet peak demand. You will have to do this across multiple hosts for high availability too. We will use the concept of *Services* to scale our application easily and manage many containers as a single entity.

Services were a new concept in Docker 1.12. They work with swarms and are intended for long-running containers.

You will perform this procedure from **node1**.

Let's deploy `sleep` as a *Service* across our Docker Swarm.

```
docker service create --name sleep-app ubuntu sleep infinity
of5rxsxsmm3asx53dqcq0o29c
```

Verify that the `service create` has been received by the Swarm manager.

```
docker service ls
```

```
ID            NAME       MODE         REPLICAS   IMAGE

of5rxsxsmm3a  sleep-app  replicated   1/1        ubuntu:latest
```

The state of the service may change a couple times until it is running. The image is being downloaded from Docker Store to the other engines in the Swarm. Once the image is downloaded the container goes into a running state on one of the three nodes.

At this point it may not seem that we have done anything very differently than just running a `docker run ...`. We have again deployed a single container on a single host. The difference here is that the container has been scheduled on a swarm cluster.

Well done. You have deployed the sleep-app to your new Swarm using Docker services.

Section 4: Scale the application

Demand is crazy! Everybody loves your `sleep` app! It's time to scale out.

One of the great things about *services* is that you can scale them up and down to meet demand. In this step you'll scale the service up and then back down.

You will perform the following procedure from **node1**.

Scale the number of containers in the **sleep-app** service to 7 with the `docker service update --replicas 7 sleep-app` command. `replicas` is the term we use to describe identical containers providing the same service.

```
docker service update --replicas 7 sleep-app
```

The Swarm manager schedules so that there are 7 `sleep-app` containers in the cluster. These will be scheduled evenly across the Swarm members.

We are going to use the `docker service ps sleep-app` command. If you do this quick enough after using the `--replicas` option you can see the containers come up in real time.

```
docker service ps sleep-app
```

```
ID            NAME         IMAGE          NODE     DESIRED STATE  CURRENT STATE
ERROR  PORTS

7k0flfh2wpt1  sleep-app.1  ubuntu:latest  node1  Running        Running 9 minutes
ago

wol6bzq7xf0v  sleep-app.2  ubuntu:latest  node3  Running        Running 2 minutes
ago

id50tzzk1qbm  sleep-app.3  ubuntu:latest  node2  Running        Running 2 minutes
ago

ozj2itmio16q  sleep-app.4  ubuntu:latest  node3  Running        Running 2 minutes
ago

o4rk5aiely2o  sleep-app.5  ubuntu:latest  node2  Running        Running 2 minutes
ago

35t0eamu0rue  sleep-app.6  ubuntu:latest  node2  Running        Running 2 minutes
ago
```

```
44s8d59vr4a8  sleep-app.7  ubuntu:latest  node1  Running        Running 2 minutes
ago
```

Notice that there are now 7 containers listed. It may take a few seconds for the new containers in the service to all show as **RUNNING**. The `NODE`column tells us on which node a container is running.

Scale the service back down to just four containers with the `docker service update --replicas 4 sleep-app` command.

```
docker service update --replicas 4 sleep-app
```

Verify that the number of containers has been reduced to 4 using the `docker service ps sleep-app` command.

```
docker service ps sleep-app
```

```
ID            NAME         IMAGE          NODE     DESIRED STATE  CURRENT STATE
ERROR  PORTS
7k0flfh2wpt1  sleep-app.1  ubuntu:latest  node1  Running        Running 13 minutes
ago
wol6bzq7xf0v  sleep-app.2  ubuntu:latest  node3  Running        Running 5 minutes
ago
35t0eamu0rue  sleep-app.6  ubuntu:latest  node2  Running        Running 5 minutes
ago
44s8d59vr4a8  sleep-app.7  ubuntu:latest  node1  Running        Running 5 minutes
ago
```

You have successfully scaled a swarm service up and down.

Section 5: Drain a node and reschedule the containers

Your sleep-app has been doing amazing after hitting Reddit and HN. It's now number 1 on the App Store! You have scaled up during the holidays and down during the slow season. Now you are doing maintenance on one of your servers so you will need to gracefully take a server out of the swarm without interrupting service to your customers.

Take a look at the status of your nodes again by running `docker node ls`on **node1**.

```
docker node ls
```

```
ID                            HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
6dlewb50pj2y66q4zi3egnwbi *   node1   Ready   Active        Leader
ym6sdzrcm08s6ohqmjx9mk3dv     node3   Ready   Active
yu3hbegvwsdpy9esh9t2lr431     node2   Ready   Active
```

You will be taking **node2** out of service for maintenance.

Let's see the containers that you have running on **node2**.

```
docker ps
```

```
CONTAINER ID        IMAGE
COMMAND             CREATED             STATUS              PORTS
NAMES

4e7ea1154ea4
ubuntu@sha256:dd7808d8792c9841d0b460122f1acf0a2dd1f56404f8d1e56298048885e45535
"sleep infinity"    9 minutes ago       Up 9 minutes
sleep-app.6.35t0eamu0rueeozz0pj2xaesi
```

You can see that we have one of the slepp-app containers running here (your output might look different though).

Now let's jump back to **node1** (the Swarm manager) and take **node2** out of service. To do that, let's run `docker node ls` again.

```
docker node ls
```

```
ID                            HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS

6dlewb50pj2y66q4zi3egnwbi *   node1   Ready   Active        Leader

ym6sdzrcm08s6ohqmjx9mk3dv     node3   Ready   Active

yu3hbegvwsdpy9esh9t2lr431     node2   Ready   Active
```

We are going to take the **ID** for **node2** and run `docker node update --availability drain yournodeid`. We are using the **node2** host **ID** as input into our `drain` command. Replace yournodeid with the id of **node2**.

```
docker node update --availability drain yournodeid
```

Check the status of the nodes

```
docker node ls
```

```
ID                            HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS

6dlewb50pj2y66q4zi3egnwbi *   node1   Ready   Active        Leader

ym6sdzrcm08s6ohqmjx9mk3dv     node3   Ready   Active

yu3hbegvwsdpy9esh9t2lr431     node2   Ready   Drain
```

Node **node2** is now in the `Drain` state.

Switch back to **node2** and see what is running there by running `docker ps`.

```
docker ps
```

```
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
```

node2 does not have any containers running on it.

Lastly, check the service again on **node1** to make sure that the container were rescheduled. You should see all four containers running on the remaining two nodes.

```
docker service ps sleep-app
```

```
ID            NAME             IMAGE          NODE     DESIRED STATE  CURRENT
STATE           ERROR  PORTS

7k0flfh2wpt1  sleep-app.1      ubuntu:latest  node1  Running        Running 25
minutes ago

wol6bzq7xf0v  sleep-app.2      ubuntu:latest  node3  Running        Running 18
minutes ago

s3548wki7rlk  sleep-app.6      ubuntu:latest  node3  Running        Running 3
minutes ago

35t0eamu0rue   \_ sleep-app.6  ubuntu:latest  node2  Shutdown       Shutdown 3
minutes ago

44s8d59vr4a8  sleep-app.7      ubuntu:latest  node1  Running        Running 18
minutes ago
```

Cleaning Up

Execute the `docker service rm sleep-app` command on **node1** to remove the service called *myservice*.

```
docker service rm sleep-app
```

Execute the `docker ps` command on **node1** to get a list of running containers.

```
docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES

044bea1c2277        ubuntu              "sleep infinity"    17 minutes ago      17
minutes ag                          distracted_mayer
```

You can use the `docker kill <CONTAINER ID>` command on **node1** to kill the sleep container we started at the beginning.

```
docker kill yourcontainerid
```

Finally, let's remove node1, node2, and node3 from the Swarm. We can use the `docker swarm leave --force` command to do that.

Lets run `docker swarm leave --force` on **node1**.

```
docker swarm leave --force
```

Then, run `docker swarm leave --force` on **node2**.

```
docker swarm leave --force
```

Finally, run `docker swarm leave --force` on **node3**.

```
docker swarm leave --force
```

Congratulations! You've completed this lab. You now know how to build a swarm, deploy applications as collections of services, and scale individual services up and down.