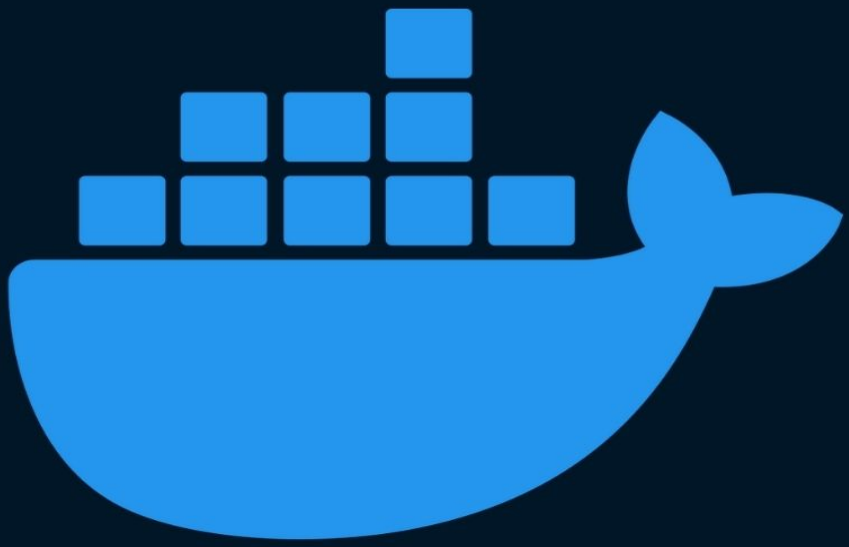
AN OPENSOURCE EBOOK

INTRODUCTION TO



docker®

Bobby Iliev

- **This version was published on October 27 2021**

This is an open-source introduction to Docker guide that will help you learn the basics of Docker and how to start using containers for your SysOps, DevOps, and Dev projects. No matter if you are a DevOps/SysOps engineer, developer, or just a Linux enthusiast, you will most likely have to use Docker at some point in your career.

The guide is suitable for anyone working as a developer, system administrator, or a DevOps engineer and wants to learn the basics of Docker.

My name is Bobby Iliev, and I have been working as a Linux DevOps Engineer since 2014. I am an avid Linux lover and supporter of the open-source movement philosophy. I am always doing that which I cannot do in order that I may learn how to do it, and I believe in sharing knowledge.

I think it's essential always to keep professional and surround yourself with good people, work hard, and be nice to everyone. You have to perform at a consistently higher level than others. That's the mark of a true professional.

For more information, please visit my blog at https://bobbyiliev.com, follow me on Twitter @bobbyiliev_ and YouTube.

This book is made possible thanks to these fantastic companies!

The Streaming Database for Real-time Analytics.

Materialize is a reactive database that delivers incremental view updates. Materialize helps developers easily build with streaming data using standard SQL.

DigitalOcean is a cloud services platform delivering the simplicity developers love and businesses trust to run production applications at scale.

It provides highly available, secure, and scalable compute, storage, and networking solutions that help developers build great software faster.

Founded in 2012 with offices in New York and Cambridge, MA, DigitalOcean offers transparent and affordable pricing, an elegant user interface, and one of the largest libraries of open source resources available.

For more information, please visit https://www.digitalocean.com or follow @digitalocean on Twitter.

If you are new to DigitalOcean, you can get a free $100 credit and spin up your own servers via this referral link here:

Free $100 Credit For DigitalOcean

The DevDojo is a resource to learn all things web development and web design. Learn on your lunch break or wake up and enjoy a cup of coffee with us to learn something new.

Join this developer community, and we can all learn together, build together, and grow together.

Join DevDojo

For more information, please visit https://www.devdojo.com or follow @thedevdojo on Twitter.

This ebook was generated by Ibis developed by Mohamed Said.

Ibis is a PHP tool that helps you write eBooks in markdown.

The cover for this ebook was created with Canva.com.

If you ever need to create a graphic, poster, invitation, logo, presentation – or anything that looks good — give Canva a go.

MIT License

It is more likely than not that **Docker** and containers are going to be part of your IT career in one way or another.

After reading this eBook, you will have a good understanding of the following:

- What is Docker
- What are containers
- What are Docker Images
- What is Docker Hub
- How to installing Docker
- How to work with Docker containers
- How to work with Docker images
- What is a Dockerfile
- How to deploy a Dockerized app
- Docker networking
- What is Docker Swarm
- How to deploy and manage a Docker Swarm Cluster

I'll be using **DigitalOcean** for all of the demos, so I would strongly encourage you to create a **DigitalOcean** account to follow along. You would learn more by doing!

To make things even better you can use my referral link to get a free $100 credit that you could use to deploy your virtual machines and test the guide yourself on a few **DigitalOcean servers**:

DigitalOcean $100 Free Credit

Once you have your account here's how to deploy your first Droplet/server:

https://www.digitalocean.com/docs/droplets/how-to/create/

I'll be using **Ubuntu 21.04** so I would recommend that you stick to the same so you could follow along.

However you can run Docker on almost any operating system including Linux, Windows, Mac, BSD and etc.

According to the official definition from the docker.com website, a container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries, and settings.

Container images become containers at runtime and in the case of Docker containers - images become containers when they run on Docker Engine. Available for both Linux and Windows-based applications, containerized software will always run the same, regardless of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging.

A **Docker Image** is just a template used to build a running Docker Container, similar to the ISO files and Virtual Machines. The containers are essentially the running instance of an image. Images are used to share containerized applications. Collections of images are stored in registries like DockerHub or private registries.

DockerHub is the default **Docker image registry** where we can store our **Docker images**. You can think of it as GitHub for Git projects.

Here's a link to the Docker Hub:

https://hub.docker.com

You can sign up for a free account. That way you could push your Docker images from your local machine to DockerHub.

Nowadays you can run Docker on Windows, Mac and of course Linux. I will only be going through the Docker installation for Linux as this is my operating system of choice.

I'll deploy an **Ubuntu server on DigitalOcean** so feel free to go ahead and do the same:

Create a Droplet DigitalOcean

Once your server is up and running, SSH to the Droplet and follow along!

If you are not sure how to SSH, you can follow the steps here:

https://www.digitalocean.com/docs/droplets/how-to/connect-with-ssh/

The installation is really straight forward, you could just run the following command, it should work on all major **Linux** distros:

```
wget -qO- https://get.docker.com | sh
```

It would do everything that's needed to install **Docker on your Linux machine**.

After that, set up Docker so that you could run it as a non-root user with the following command:

```
sudo usermod -aG docker ${USER}
```

To test **Docker** run the following:

```
docker version
```

To get some more information about your Docker Engine, you can run the following command:

```
docker info
```

With the `docker info` command, we can see how many running containers that we've got and some server information.

The output that you would get from the docker version command should look something like this:

In case you would like to install Docker on your Windows PC or on your Mac, you could visit the official Docker documentation here:

https://docs.docker.com/docker-for-windows/install/

And:

https://docs.docker.com/docker-for-mac/install/

That is pretty much it! Now you have Docker running on your machine!

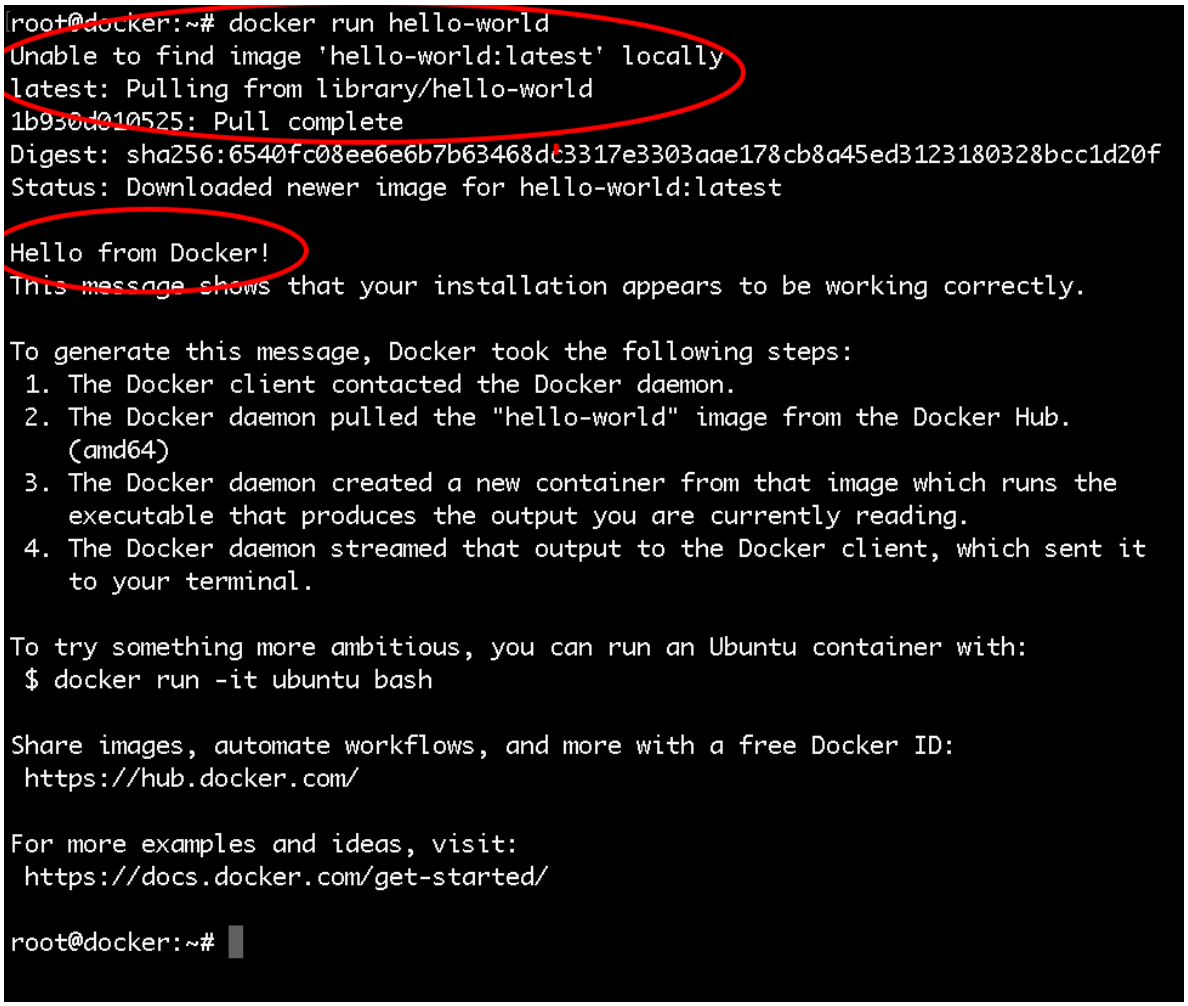
Now we are ready to start working with containers! We will pull a **Docker image** from the **DockerHub**, we will run a container, stop it, destroy it and more!

Once you have your **Ubuntu Droplet** ready, ssh to the server and follow along!

So let's run our first Docker container! To do that you just need to run the following command:

```
docker run hello-world
```

You will get the following output:



We just ran a container based on the **hello-world Docker Image**, as we did not have the image locally, docker pulled the image from the DockerHub and then used that image to run the container. All that happened was: the **container ran**, printed some text on the screen and then exited.

Then to see some information about the running and the stopped containers run:

```
docker ps -a
```

You will see the following information for your **hello-world container** that you just ran:

```
root@docker:~# docker ps -a
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS                      PORTS
NAMES
62d360207d08        hello-world         "/hello"            5
minutes ago       Exited (0) 5 minutes ago
focused_cartwright
```

In order to list the locally available Docker images on your host run the following command:

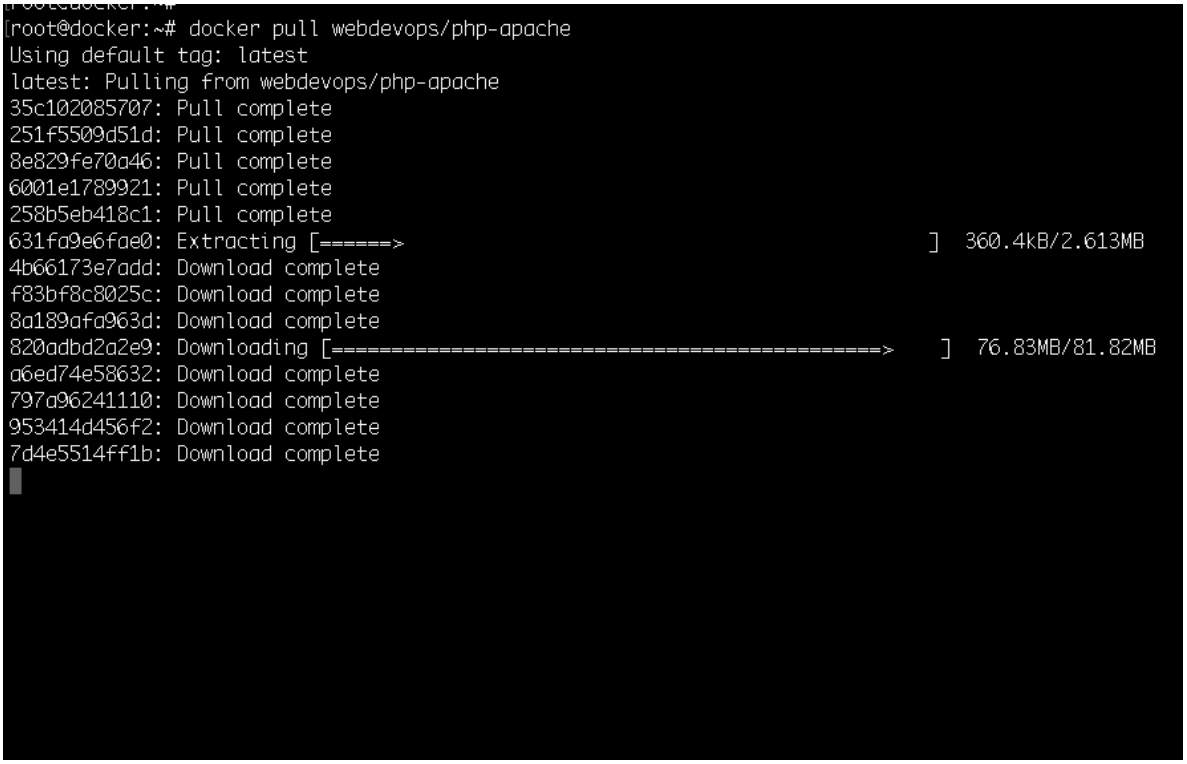
```
docker images
```

Let's run a more useful container like an **Apache** container for example.

First, we can pull the image from the docker hub with the **docker pull command**:

```
docker pull webdevops/php-apache
```

You will see the following output:



Then we can get the image ID with the docker images command:

```
docker images
```

The output would look like this:

```
root@docker:~# docker images
REPOSITORY              TAG                 IMAGE ID            CREATED             SIZE
webdevops/php-apache    latest              fd4f7e58ef4b        8 hours ago         531MB
hello-world             latest              fce289e99eb9        7 months ago        1.84kB
root@docker:~#
```

Note, you do not necessarily need to pull the image, this is just for demo purposes. When running the `docker run` command, if the image is not available locally, it will automatically be pulled from Docker Hub.

After that we can use the **docker run** command to spin up a new container:

```
docker run -d -p 80:80 IMAGE_ID
```

Quick rundown of the arguments that I've used:

- `-d`: it specifies that I want to run the container in the background. That way when you close your terminal the container would remain running.

- `-p 80:80`: this means that the traffic from the host on port 80 would be forwarded to the container. That way you could access the Apache instance which is running inside your docker container directly via your browser.

With the docker info command now we can see that we have 1 running container.

And with the `docker ps` command we could see some useful information about the container like the container ID, when the container was started, etc.:

```
root@docker:~# docker ps
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS
NAMES
7dd1d512b50e        fd4f7e58ef4b        "/entrypoint supervi…"
About a minute ago   Up About a minute   443/tcp,
0.0.0.0:80->80/tcp, 9000/tcp   pedantic_murdock
```

Then you can stop the running container with the docker stop command followed by the container ID:

```
docker stop CONTAINER_ID
```

If you need to, you can start the container again:

```
docker start CONTAINER_ID
```

In order to restart the container you can use the following:

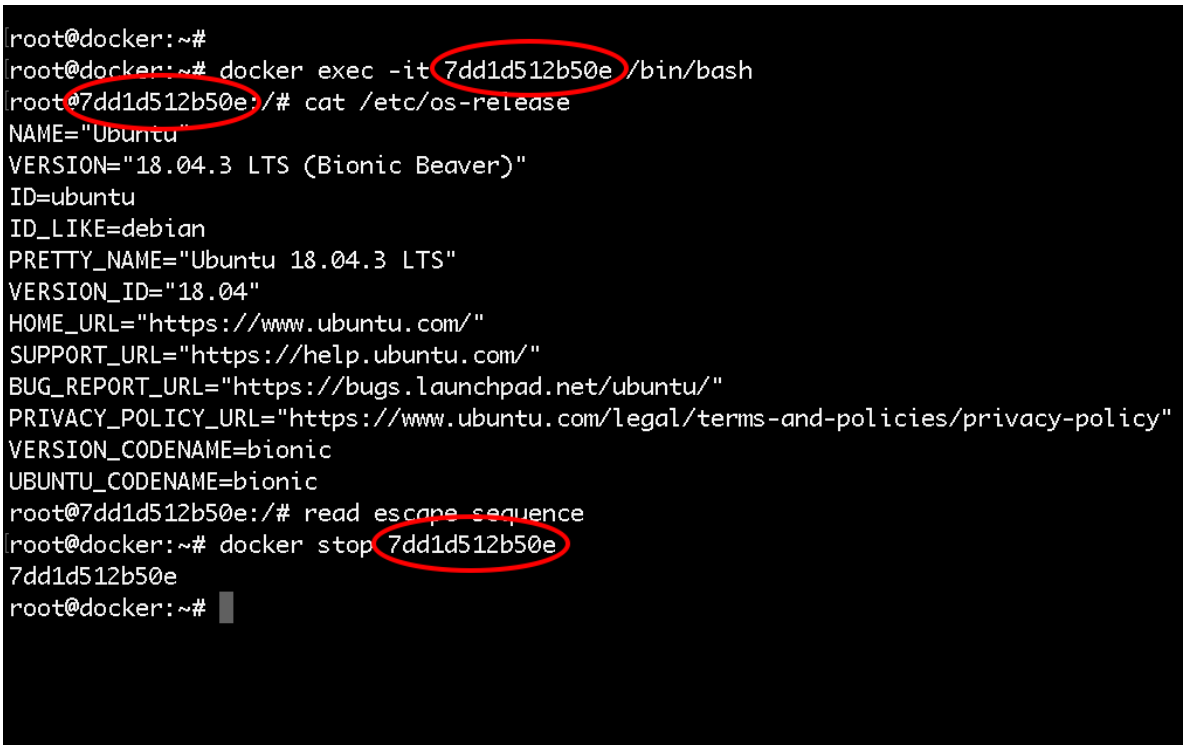
```
docker restart CONTAINER_ID
```

If you need to attach to the container and run some commands inside the container use the `docker exec` command:

```
docker exec -it CONTAINER_ID /bin/bash
```

That way you will get to a **bash shell** in the container and execute some commands inside the container itself.

Then, to detach from the interactive shell, press `CTRL+PQ`. That way you will not stop the container but just detach it from the interactive shell.



To delete the container, first make sure that the container is not running and then run:

```
docker rm CONTAINER_ID
```

If you would like to delete the container and the image all together, just run:

```
docker rmi IMAGE_ID
```

With that you now know how to pull Docker images from the **Docker Hub**, run, stop, start and even attach to Docker containers!

We are ready to learn how to work with **Docker images!**

A Docker Image is just a template used to build a running Docker Container, similar to the ISO files and Virtual Machines. The containers are essentially the running instance of an image. Images are used to share containerized applications. Collections of images are stored in registries like DockerHub or private registries.

The `docker run` command downloads and runs images at the same time. But we could also only download images if we wanted to with the docker pull command. For example:

```
docker pull ubuntu
```

Or if you want to get a specific version you could also do that with:

```
docker pull ubuntu:14.04
```

Then to list all of your images use the docker images command:

```
docker images
```

You would get a similar output to:

```
root@docker:~# docker images
REPOSITORY              TAG                 IMAGE ID            CREATED             SIZE
webdevops/php-apache    latest              fd4f7e58ef4b        8 hours ago         531MB
ubuntu                  latest              a2a15febcdf3        2 days ago          64.2MB
ubuntu                  14.04               2c5e00d77a67        3 months ago        188MB
hello-world             latest              fce289e99eb9        7 months ago        1.84kB
root@docker:~#
```

The images are stored locally on your docker host machine.

To take a look a the docker hub, go to: https://hub.docker.com and you would be able to see where the images were just downloaded from.

For example, here's a link to the **Ubuntu image** that we've just downloaded:

https://hub.docker.com/_/ubuntu

There you could find some useful information.

As Ubuntu 14.04 is really outdated, to delete the image use the `docker rmi`

command:

```
docker rmi ubuntu:14.04
```

One of the ways of modifying images is with ad-hoc commands. For example just start your ubuntu container.

```
docker run -d -p 80:80 IMAGE_ID
```

After that to attach to your running container you can run:

```
docker exec -it container_name /bin/bash
```

Install whatever packages needed then exit the container just press CTRL+P+Q.

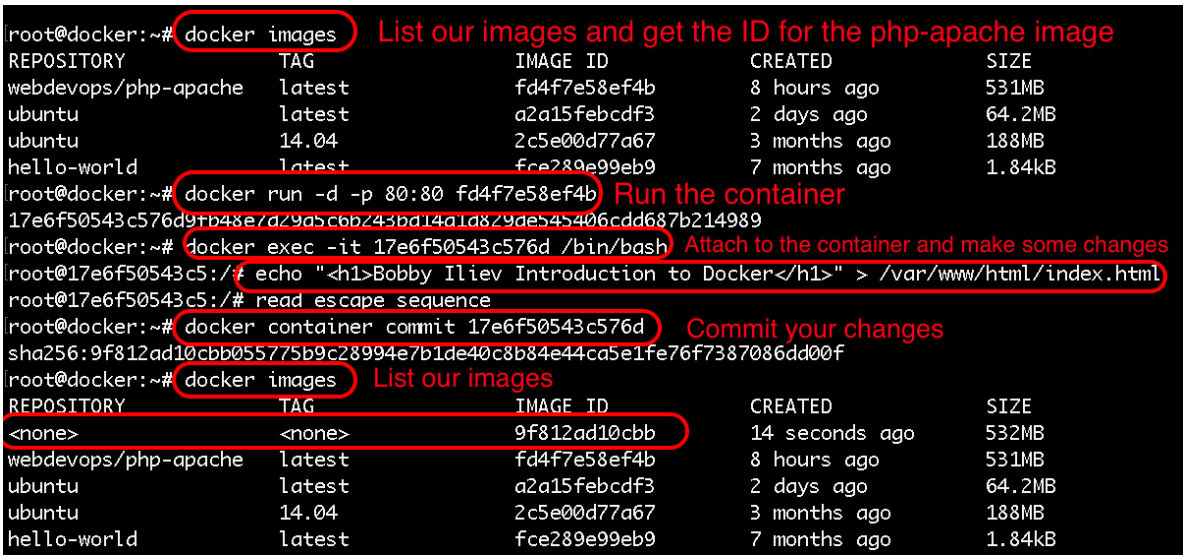
To save your changes run the following:

```
docker container commit ID_HERE
```

Then list your images and note your image ID:

```
docker images ls
```

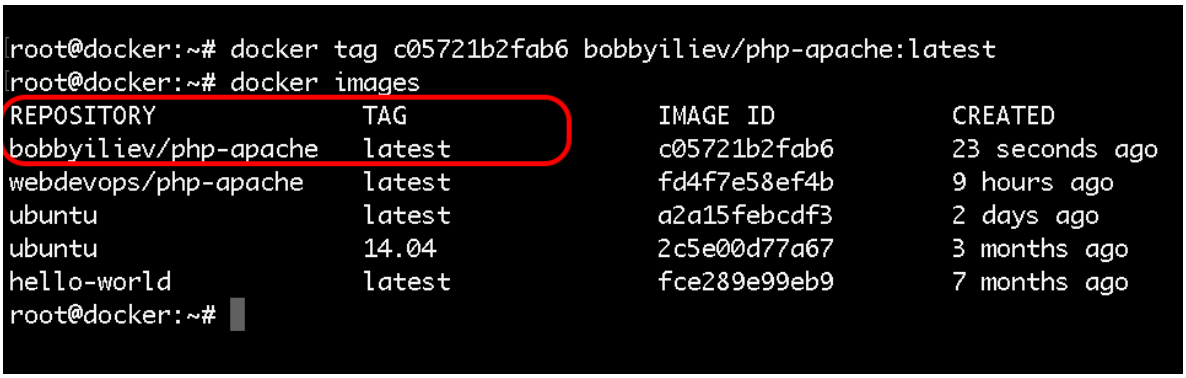
The process would look as follows:



As you would notice your newly created image would not have a name nor a tag, so in order to tag your image run:

```
docker tag IMAGE_ID YOUR_TAG
```

Now if you list your images you would see the following output:



Now that we have our new image locally, let's see how we could push that new image to DockerHub.

For that you would need a Docker Hub account first. Then once you have your account ready, in order to authenticate, run the following command:

```
docker login
```

Then push your image to the **Docker Hub**:

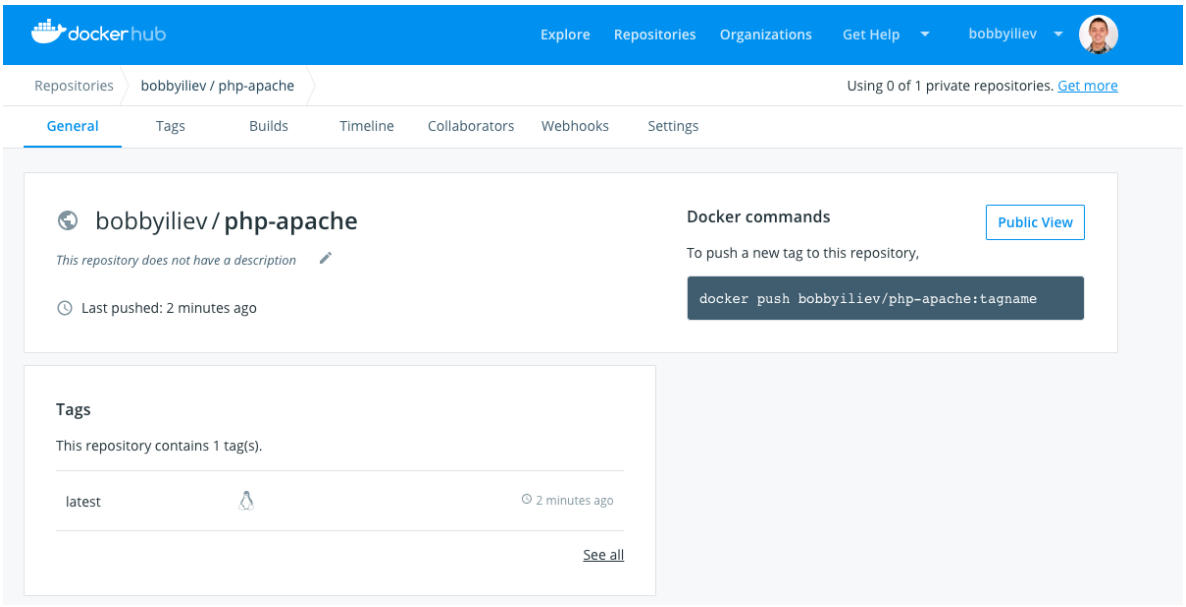
```
docker push your-docker-user/name-of-image-here
```

The output would look like this:

```
Login Succeeded
root@docker:~# docker images
REPOSITORY              TAG                 IMAGE ID
bobbyiliev/php-apache   latest              c05721b2fab6
webdevops/php-apache    latest              fd4f7e58ef4b
ubuntu                  latest              a2a15febcdf3
ubuntu                  14.04               2c5e00d77a67
hello-world             latest              fce289e99eb9
root@docker:~# docker push bobbyiliev/php-apache
The push refers to repository [docker.io/bobbyiliev/php-apache]
868169b4b463: Pushed
7932a6d7fb8b: Mounted from webdevops/php-apache
dd8c9ac68361: Mounted from webdevops/php-apache
192e1c6f533d: Mounted from webdevops/php-apache
7366b208287e: Mounted from webdevops/php-apache
d8e58967ec00: Mounted from webdevops/php-apache
0fb9f5e3bca8: Mounted from webdevops/php-apache
4bce5b11e428: Mounted from webdevops/php-apache
7479eb10d8dd: Mounted from webdevops/php-apache
df021786f442: Waiting
7576a954eea3: Waiting
122be11ab4a2: Waiting
7beb13bce073: Waiting
f7eae43028b3: Waiting
6cebf3abed5f: Waiting
```

After that you should be able to see your docker image in your docker hub account, in my case it would be here:

https://cloud.docker.com/repository/docker/bobbyiliev/php-apache


docker hub
Explore
Repositories
Organizations
Get Help
bobbyiliev
Repositories
bobbyiliev / php-apache
Using 0 of 1 private repositories. Get more
General
Tags
Builds
Timeline
Collaborators
Webhooks
Settings
bobbyiliev / php-apache
This repository does not have a description
Last pushed: 2 minutes ago
Docker commands
Public View
To push a new tag to this repository,
docker push bobbyiliev/php-apache:tagname
Tags
This repository contains 1 tag(s).
latest
2 minutes ago
See all

We will go the Dockerfile a bit more in depth in the next blog post, for this demo we will only use a simple Dockerfile just as an example:

Create a file called `Dockerfile` and add the following content:

```
FROM alpine
RUN apk update
```

All that this `Dockerfile` does is to update the base Alpine image.

To build the image run:

```
docker image build -t alpine-updated:v0.1 .
```

Then you could again list your image and push the new image to the **Docker Hub**!

Once you've read this post, make sure to test your knowledge with this Docker Images Quiz:

https://quizapi.io/predefined-quizzes/common-docker-images-questions

Now that you know how to pull, modify, and push **Docker images**, we are ready to learn more about the `Dockerfile` and how to use it!

A **Dockerfile** is basically a text file that contains all of the required commands to build a certain **Docker image**.

The **Dockerfile** reference page:

https://docs.docker.com/engine/reference/builder/

It lists the various commands and format details for Dockerfiles.

Here's a really basic example of how to create a `Dockerfile` and add our source code to an image.

First, I have a simple Hello world `index.html` file in my current directory that I would add to the container with the following content:

```
<h1>Hello World - Bobby Iliev</h1>
```

And I also have a Dockerfile with the following content:

```
FROM webdevops/php-apache-dev
MAINTAINER Bobby I.
COPY . /var/www/html
WORKDIR /var/www/html
EXPOSE 8080
```

Here is a screenshot of my current directory and the content of the files:

```
root@docker:~/demo# ls -l
total 8
-rw-r--r-- 1 root root 135 Aug 17 10:40 Dockerfile
-rw-r--r-- 1 root root  35 Aug 17 10:40 index.html
root@docker:~/demo#
root@docker:~/demo#
root@docker:~/demo#
root@docker:~/demo# cat index.html
<h1>Hello World - Bobby Iliev</h1>
root@docker:~/demo#
root@docker:~/demo#
root@docker:~/demo#
root@docker:~/demo# cat Dockerfile
FROM webdevops/php-apache-dev
MAINTAINER Bobby I.
COPY . /var/www/html
WORKDIR /var/www/html
RUN /etc/init.d/apache2 start
EXPOSE 8080
root@docker:~/demo#
root@docker:~/demo#
root@docker:~/demo#
```

Here is a quick rundown of the Dockerfile:

- `FROM`: The image that we would use as a ground
- `MAINTAINER`: The person who would be maintaining the image
- `COPY`: Copy some files in the image
- `WORKDIR`: The directory where you want to run your commands on start

- `EXPOSE`: Specify a port that you would like to access the container on

Now in order to build a new image from our `Dockerfile`, we need to use the docker build command. The syntax of the docker build command is the following:

```
docker build [OPTIONS] PATH | URL | -
```

The exact command that we need to run is this one:

```
docker build -f Dockerfile -t your_user_name/php-apache-dev .
```

After the built is complete you can list your images with the docker images command and also run it:

```
docker run -d -p 8080:80 your_user_name/php-apache-dev
```

And again just like we did in the last step, we can go ahead and publish our image:

```
docker login

docker push your-docker-user/name-of-image-here
```

Then you will be able to see your new image in your Docker Hub account (https://hub.docker.com) you can pull from the hub directly:

```
docker pull your-docker-user/name-of-image-here
```

For more information on the docker build make sure to check out the official documentation here:

https://docs.docker.com/engine/reference/commandline/build/

Once you've read this post, make sure to test your knowledge with this **Dockerfile quiz**:

https://quizapi.io/predefined-quizzes/basic-dockerfile-quiz

This is a really basic example, you could go above and beyond with your Dockerfiles!

Now you know how to write a Dockerfile, how to build a new image from a Dockerfile using the docker build command!

In the next step we will learn how to set up and work with the **Docker Swarm** mode!

Docker comes with a pluggable networking system. There are multiple plugins that you could use by default:

- `bridge`: The default Docker network driver. This is suitable for standalone containers that need to communicate with each other.
- `host`: This driver removes the network isolation between the container and the host. This is suitable for standalone containers which use the host network directly.
- `overlay`: Overlay allows you to connect multiple Docker daemons. This enables you to run Docker swarm services by allowing them to communicate with each other.
- `none`: Disables all networking.

In order to list the currently available Docker networks you can use the following command:

```
docker network list
```

You would get the following output:

```
NETWORK ID     NAME      DRIVER    SCOPE
3194399146e4   bridge    bridge    local
cf7f50175100   host      host      local
590fb3abc0e1   none      null      local
```

As you can see, we have 3 networks available out of the box already with 3 of the network drivers that we've discussed above.

To create a new Docker network with the default `bridge` driver you can run the following command:

```
docker network create myNewNetwork
```

The above command would create a new network with the name of `myNewNetwork`.

You can also specify a different driver by adding the `--driver=DRIVER_NAME` flag.

If you want to create a Docker network with a specific range, you can do that by adding the `--subnet=` flag followed by the subnet that you want to use.

In order to get some information for an existing Docker network like the driver that is being used, the subnet, the containers attached to that network, you can use the `docker network inspect` command as follows:

```
docker network inspect myNewNetwork
```

The output of the command would be in JSON by default.

You can use the `docker inspect` command to inspect other Docker objects like containers, images and etc.

To practice what you've just learned, let's create two containers and add them to a Docker network so that they could communicate with each other using their container names.

Here is a quick example of a `bridge` network:

- First start by creating two containers:

```
docker run -d --name web1  -p 8001:80 eboraas/apache-php
docker run -d --name web2  -p 8002:80 eboraas/apache-php
```

It is very **important** to **explicitly specify a name** with `--name` for your containers otherwise I've noticed that it would not work with the random names that Docker assigns to your containers.

- Once the two containers are up and running, create a new network:

```
docker network create myNetwork
```

- After that connect your containers to the network:

```
docker network connect myNetwork web1
docker network connect myNetwork web2
```

- Check if your containers are part of the new network:

```
docker network inspect myNetwork
```

- Then test the connection:

```
docker exec -ti web1 ping web2
```

Again, keep in mind that it is quite important to explicitly specify names for your containers otherwise this would not work. I figured this out after spending a few hours trying to figure it out.

For more information about the power of the Docker network, make sure to check the official documentation here.

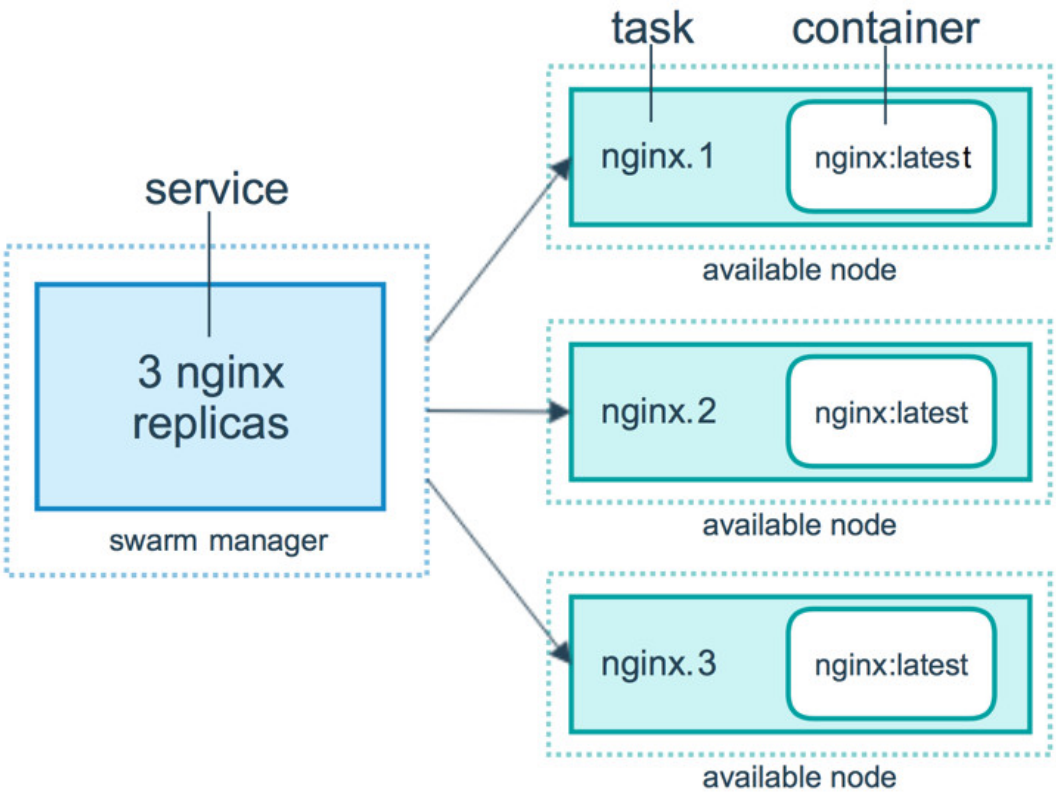
According to the official **Docker** docs, a swarm is a group of machines that are running **Docker** and joined into a cluster. If you are running a **Docker swarm** your commands would be executed on a cluster by a swarm manager. The machines in a swarm can be physical or virtual. After joining a swarm, they are referred to as nodes. I would do a quick demo shortly on my **DigitalOcean** account!

The **Docker Swarm** consists of **manager nodes** and **worker nodes**.

The manager nodes dispatch tasks to the worker nodes and on the other side Worker nodes just execute those tasks. For High Availability, it is recommended to have **3** or **5** manager nodes.

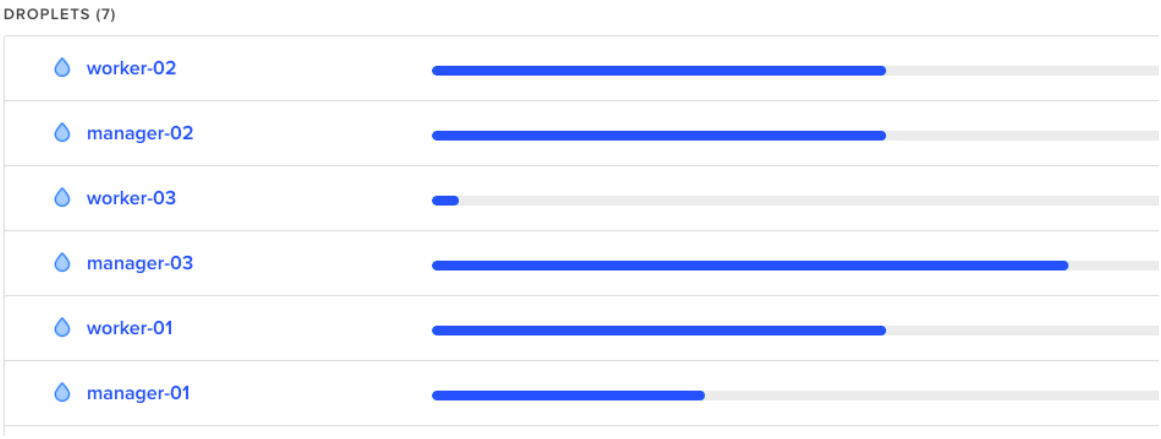
To deploy an application image when Docker Engine is in swarm mode, you have create a service. A service is a group of containers of the same `image:tag`. Services make it simple to scale your application.

In order to have **Docker services**, you must first have your **Docker swarm** and nodes ready.



I'll do a really quick demo on how to build a **Docker swarm with 3 managers and 3 workers**.

For that I'm going to deploy 6 droplets on DigitalOcean:



Then once you've got that ready, **install docker** just as we did in the Introduction to Docker Part 1 and then just follow the steps here:

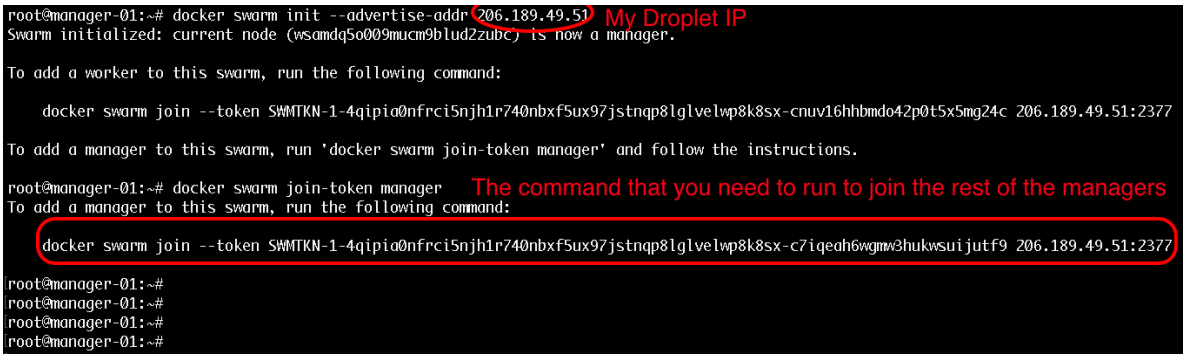
Initialize the docker swarm on your first manager node:

```
docker swarm init --advertise-addr your_dorplet_ip_here
```

Then to get the command that you need to join the rest of the managers simply run this:

```
docker swarm join-token manager
```

Note: This would provide you with the exact command that you need to run on the rest of the swarm manager nodes. Example:



To get the command that you need for joining workers just run:

```
docker swarm join-token worker
```

The command for workers would be pretty similar to the command for join managers but the token would be a bit different.

The output that you would get when joining a manager would look like this:

Then once you have your join commands, **ssh to the rest of your nodes and join them** as workers and managers accordingly.

After you've run the join commands on all of your workers and managers, in order to get some information for your cluster status you could use these commands:

- To list all of the available nodes run:

```
docker node ls
```

Note: This command can only be run from a **swarm manager**!Output:

```
root@manager-01:~# docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS      ENGINE VE
RSION
wsamdq5o009mucm9blud2zubc *   manager-01          Ready               Active              Leader              19.03.1
tu6qjtp2snv0aoohc97s2a6bo     manager-02          Ready               Active              Reachable           19.03.1
q76nlmjz8ji3tibnun6idbqb2     manager-03          Ready               Active              Reachable           19.03.1
mz203d95uc21ut6mpm7hujovu     worker-01           Ready               Active                                  19.03.1
bappqrc7xj8iem5za8eaps60o     worker-02           Ready               Active                                  19.03.1
o3l531dwbk07utrwzxqe884bf     worker-03           Ready               Active                                  19.03.1
root@manager-01:~#
```

- To get information for the current state run:

```
docker info
```

Output:

```
root@manager-01:~# docker info
Client:
 Debug Mode: false

Server:
 Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
 Images: 0
 Server Version: 19.03.1
 Storage Driver: overlay2
  Backing Filesystem: extfs
  Supports d_type: true
  Native Overlay Diff: true
 Logging Driver: json-file
 Cgroup Driver: cgroupfs
 Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local
 Swarm: active
  NodeID: wsamdq5o009mucm9blud2zubc
  Is Manager: true
  ClusterID: 2sj7v47f4izkhztyzdmvoc30i
  Managers: 3
  Nodes: 6
  Default Address Pool: 10.0.0.0/8
  SubnetSize: 24
  Data Path Port: 4789
  Orchestration:
   Task History Retention Limit: 5
  Raft:
```

To promote a worker to a manager run the following from **one** of your manager nodes:

```
docker node promote node_id_here
```

Also note that each manager also acts as a worker, so from your docker info output you should see 6 workers and 3 manager nodes.

In order to create a service you need to use the following command:

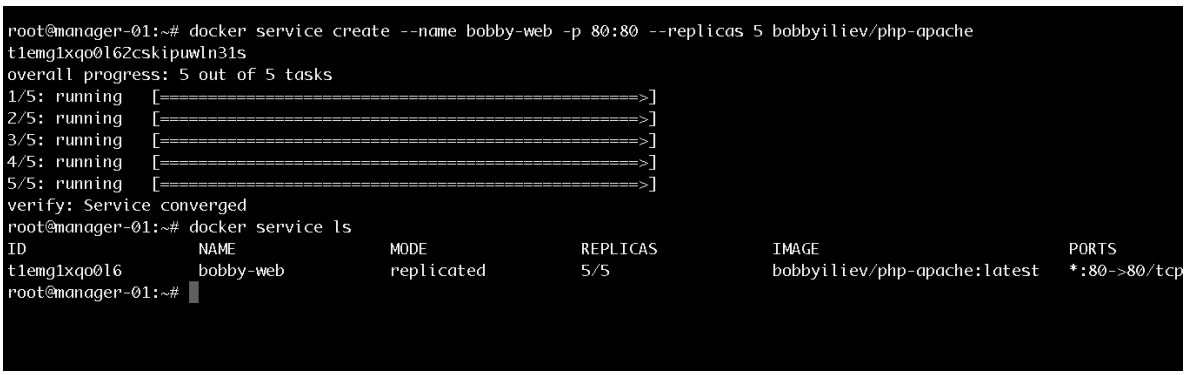
```
docker service create --name bobby-web -p 80:80 --replicas 5
bobbyiliev/php-apache
```

Note that I already have my bobbyiliev/php-apache image pushed to the Docker hub as described in the previous blog posts.

To get a list of your services run:

```
docker service ls
```

Output:



Then in order to get a list of the running containers you need to use the following command:

```
docker services ps name_of_your_service_here
```

Output:

```
root@manager-01:~# docker service ps bobby-web
ID                  NAME                IMAGE                          NODE              DESIRED STATE       CURRENT STATE
OR              PORTS
ivlg5dnrtti4        bobby-web.1         bobbyiliev/php-apache:latest   manager-03        Running             Running 2 minutes ago

i0jmepncpdz5        bobby-web.2         bobbyiliev/php-apache:latest   worker-02         Running             Running about a minute ago

7rvnlx7lkefn        bobby-web.3         bobbyiliev/php-apache:latest   worker-03         Running             Running about a minute ago

loha42ahdtfj        bobby-web.4         bobbyiliev/php-apache:latest   manager-01        Running             Running about a minute ago

oowdp9glb7iz        bobby-web.5         bobbyiliev/php-apache:latest   manager-02        Running             Running about a minute ago

root@manager-01:~#
```

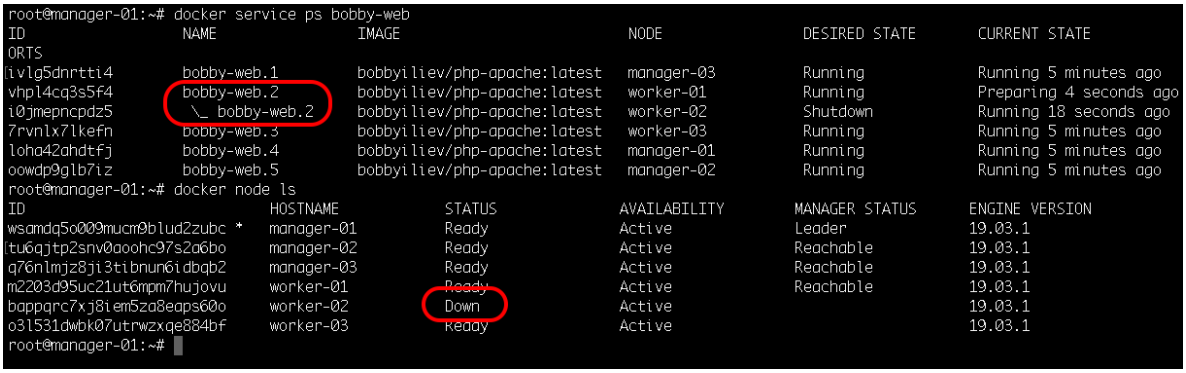
Then you can visit the IP address of any of your nodes and you should be able to see the service! We can basically visit any node from the swarm and we will still get the to service.

We could try shutting down one of the nodes and see how the swarm would automatically spin up a new process on another node so that it matches the desired state of 5 replicas.

To do that go to your **DigitalOcean** control panel and hit the power off button for one of your Droplets. Then head back to your terminal and run:

```
docker services ps name_of_your_service_here
```

Output:

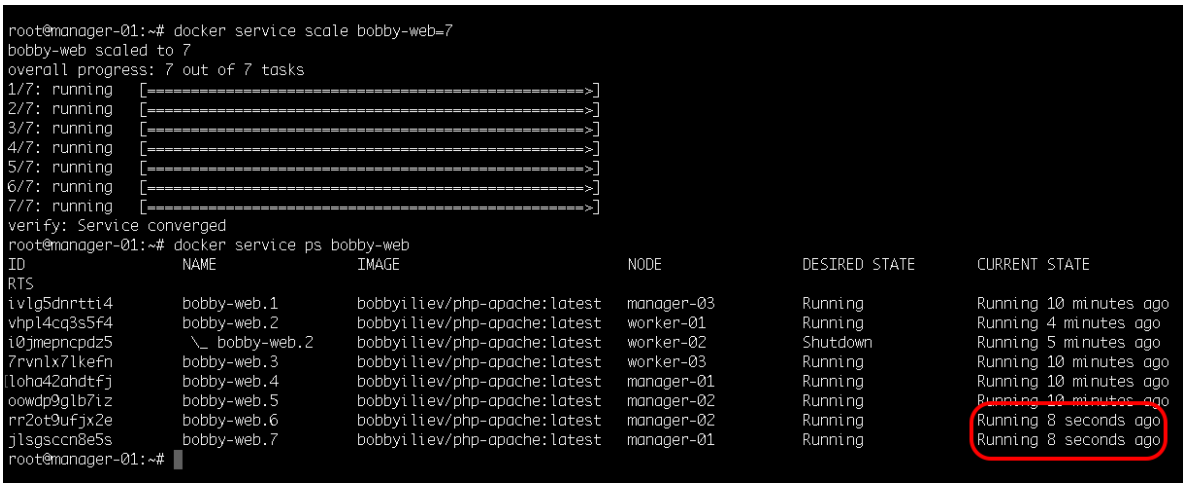


In the screenshot above, you can see how I've shutdown the droplet called worker-2 and how the replica bobby-web.2 was instantly started again on another node called worker-01 to match the desired state of 5 replicas.

To add more replicas run:

```
docker service scale name_of_your_service_here=7
```

Output:



This would automatically spin up 2 more containers, you can check this with the docker service ps command:

```
docker service ps name_of_your_service_here
```

Then as a test try starting the node that we've shutdown and check if it picked up any tasks?

Tip: Bringing new nodes to the cluster does not automatically distribute running tasks.

In order to delete a service, all you need to do is to run the following command:

```
docker service rm name_of_your_service
```

Output:

Now you know how to initialize and scale a docker swarm cluster! For more information make sure to go through the official Docker documentation here.

Once you've read this post, make sure to test your knowledge with this **Docker Swarm Quiz**:

https://quizapi.io/predefined-quizzes/common-docker-swarm-interview-questions

Congratulations! You have just completed the Docker basics eBook! I hope that it was helpful and you've managed to learn some cool new things about Docker!

If you found this helpful, be sure to star the project on GitHub!

If you have any suggestions for improvements, make sure to contribute pull requests or open issues.

In this introduction to Docker eBook, we just covered the basics, but you still have enough under your belt to start working with Docker containers and images!

As a next step make sure to spin up a few servers, install Docker and play around with all of the commands that you've learnt from this eBook!

In case that this eBook inspired you to contribute to some fantastic open-source project, make sure to tweet about it and tag @bobbyiliev_ so that we could check it out!

Congrats again on completing this eBook!

Some other opensource eBooks that you might find helpful are:

- Introduction to Git and GitHub
- Introduction to Bash Scripting
- Introduction to SQL