Table of Contents

[Docker Basics 3](#_Toc118113360)

[Installation 3](#_Toc118113361)

[Orientation 3](#_Toc118113362)

[Docker concepts 3](#_Toc118113363)

[Images and containers 3](#_Toc118113364)

[Containers and virtual machines 4](#_Toc118113365)

[Docker commands 4](#_Toc118113366)

[Terminology 6](#_Toc118113367)

[Docker Images 6](#_Toc118113368)

[Our First Image 7](#_Toc118113369)

[Another Example: Py Flask + Redis 10](#_Toc118113370)

[Containers 11](#_Toc118113371)

[To stop the container on Windows 10 12](#_Toc118113372)

[Now, run app in the background 12](#_Toc118113373)

[Execute commands on the container 12](#_Toc118113374)

[Share your image 12](#_Toc118113375)

[Log on to docker 12](#_Toc118113376)

[Tag the image 12](#_Toc118113377)

[Publish the image 13](#_Toc118113378)

[Pull and run the image from the remote repository 13](#_Toc118113379)

[Delete the image and container locally and then Pull and run from docker hub 13](#_Toc118113380)

[Services 13](#_Toc118113381)

[Run your new load-balanced app 14](#_Toc118113382)

[Scale the app 15](#_Toc118113383)

[Take down the app and the swarm 15](#_Toc118113384)

[Swarms 15](#_Toc118113385)

[Understanding Swarm clusters 15](#_Toc118113386)

[Setup your swarm 16](#_Toc118113387)

[Deploy your app on the swarm cluster 17](#_Toc118113388)

[Accessing your cluster 18](#_Toc118113389)

[Cleanup and reboot 18](#_Toc118113390)

[Stacks and swarms 18](#_Toc118113391)

[Unsetting docker-machine shell variable settings 18](#_Toc118113392)

[Restarting Docker machines 19](#_Toc118113393)

[Stacks 19](#_Toc118113394)

[Add a new service and redeploy 19](#_Toc118113395)

[Persist the data 22](#_Toc118113396)

[Using Multiple Containers with Docker Compose 26](#_Toc118113397)

[MySQL and .NET 26](#_Toc118113398)

[PostgreSQL and .NET 26](#_Toc118113399)

[Better productivity with Docker Compose 28](#_Toc118113400)

[Detached mode 29](#_Toc118113401)

[Docker Multi Stage Build 30](#_Toc118113402)

[Before multi-stage builds 30](#_Toc118113403)

[How to Use Docker Multistage Builds 30](#_Toc118113404)

[Docker Deployment to Azure 32](#_Toc118113405)

# Docker Basics

<https://www.freecodecamp.org/news/learn-docker-and-kubernetes-hands-on-course/>

<https://stackify.com/docker-tutorial/>

<https://www.tutorialspoint.com/docker/index.htm>

https://docker-curriculum.com/

## Installation

Docker Desktop for Windows: <https://docs.docker.com/docker-for-windows/install/>

Docker on Windows 10 Home:

* Install Oracle Virtual Box and then create a new VM using docker-machine.
* <https://docs.docker.com/machine/get-started/>
* <https://docs.docker.com/machine/drivers/virtualbox/>

Docker Desktop for Mac: <https://docs.docker.com/docker-for-mac/install/>

Docker on Linux: <https://runnable.com/docker/install-docker-on-linux>

Gitbash: <https://git-scm.com/downloads>

## Orientation

### Docker concepts

Docker is a platform for developers and sysadmins to **develop, deploy, and run** applications with containers. The use of Linux containers to deploy applications is called containerization. Containers are not new, but their use for easily deploying applications is.

Containerization is increasingly popular because containers are:

* Flexible: Even the most complex applications can be containerized.
* Lightweight: Containers leverage and share the host kernel.
* Interchangeable: You can deploy updates and upgrades on-the-fly.
* Portable: You can build locally, deploy to the cloud, and run anywhere.
* Scalable: You can increase and automatically distribute container replicas.
* Stackable: You can stack services vertically and on-the-fly.

### Images and containers

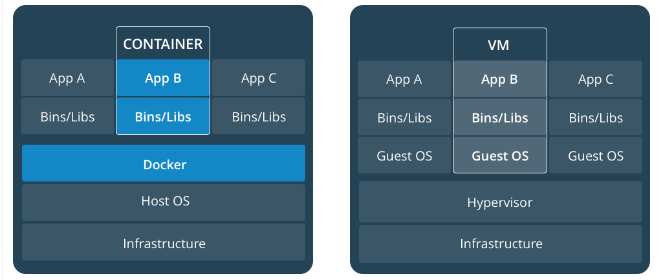
A container is launched by running an image. An **image** is an executable package that includes everything needed to run an application--the code, a runtime, libraries, environment variables, and configuration files.

A **container** is a runtime instance of an image--what the image becomes in memory when executed (that is, an image with state, or a user process). You can see a list of your running containers with the command, docker ps, just as you would in Linux.

### Containers and virtual machines

A **container** runs natively on Linux and shares the kernel of the host machine with other containers. It runs a discrete process, taking no more memory than any other executable, making it lightweight.

By contrast, a **virtual machine** (VM) runs a full-blown “guest” operating system with virtual access to host resources through a hypervisor. In general, VMs provide an environment with more resources than most applications need.



### Docker commands

## List Docker CLI commands

docker

docker container --help

## Display Docker version and info

docker --version

docker version

docker info

## Execute Docker image

docker run hello-world

## List Docker images

docker image ls

## List Docker containers (running, all, all in quiet mode)

docker container ls

docker container ls --all

docker container ls -aq

**More**:

docker run busybox

Wait, nothing happened! Is that a bug? Well, no. Behind the scenes, a lot of stuff happened. When you call run, the Docker client finds the image (busybox in this case), loads up the container and then runs a command in that container. When we run docker run busybox, we didn't provide a command, so the container booted up, ran an empty command and then exited. Well, yeah - kind of a bummer. Let's try something more exciting.

docker run busybox echo "hello from busybox"

Nice - finally we see some output. In this case, the Docker client dutifully ran the echo command in our busybox container and then exited it. If you've noticed, all of that happened pretty quickly. Imagine booting up a virtual machine, running a command and then killing it. Now you know why they say containers are fast! Ok, now it's time to see the docker ps command. The docker ps command shows you all containers that are currently running.

docker ps

Since no containers are running, we see a blank line. Let's try a more useful variant: docker ps -a

docker ps -a

So, what we see above is a list of all containers that we ran. Do notice that the STATUS column shows that these containers exited a few minutes ago.

You're probably wondering if there is a way to run more than just one command in a container. Let's try that now:

docker run -it busybox sh

/ # ls

bin dev etc home proc root sys tmp usr var

/ # uptime

Running the run command with the -it flags attaches us to an interactive tty in the container. Now we can run as many commands in the container as we want. Take some time to run your favorite commands.

Before we move ahead though, let's quickly talk about deleting containers. We saw above that we can still see remnants of the container even after we've exited by running docker ps -a. Throughout this tutorial, you'll run docker run multiple times and leaving stray containers will eat up disk space. Hence, as a rule of thumb, I clean up containers once I'm done with them. To do that, you can run the docker rm command. Just copy the container IDs from above and paste them alongside the command.

docker rm 305297d7a235 ff0a5c3750b9

On deletion, you should see the IDs echoed back to you. If you have a bunch of containers to delete in one go, copy-pasting IDs can be tedious. In that case, you can simply run -

docker rm $(docker ps -a -q -f status=exited)

This command deletes all containers that have a status of exited. In case you're wondering, the -q flag, only returns the numeric IDs and -f filters output based on conditions provided. One last thing that'll be useful is the --rm flag that can be passed to docker run which automatically deletes the container once it's exited from. For one off docker runs, --rm flag is very useful.

In later versions of Docker, the docker container prune command can be used to achieve the same effect.

docker container prune

Lastly, you can also delete images that you no longer need by running docker rmi.

## Terminology

In the last section, we used a lot of Docker-specific jargon which might be confusing to some. So before we go further, let me clarify some terminology that is used frequently in the Docker ecosystem.

* Images - The blueprints of our application which form the basis of containers. In the demo above, we used the docker pull command to download the **busybox** image.
* Containers - Created from Docker images and run the actual application. We create a container using docker run which we did using the busybox image that we downloaded. A list of running containers can be seen using the docker ps command.
* Docker Daemon - The background service running on the host that manages building, running and distributing Docker containers. The daemon is the process that runs in the operating system which clients talk to.
* Docker Client - The command line tool that allows the user to interact with the daemon. More generally, there can be other forms of clients too - such as [Kitematic](https://kitematic.com/) which provide a GUI to the users.
* Docker Hub - A [registry](https://hub.docker.com/explore/) of Docker images. You can think of the registry as a directory of all available Docker images. If required, one can host their own Docker registries and can use them for pulling images.

## Docker Images

We've looked at images before, but in this section we'll dive deeper into what Docker images are and build our own image! Lastly, we'll also use that image to run our application locally and finally share it with our friends! Excited? Great! Let's get started.

Docker images are the basis of containers. In the previous example, we **pulled** the Busybox image from the registry and asked the Docker client to run a container **based** on that image. To see the list of images that are available locally, use the docker images command.

docker images

The above gives a list of images that I've pulled from the registry, along with ones that I've created myself (we'll shortly see how). The TAG refers to a particular snapshot of the image and the IMAGE ID is the corresponding unique identifier for that image.

For simplicity, you can think of an image akin to a git repository - images can be [committed](https://docs.docker.com/engine/reference/commandline/commit/) with changes and have multiple versions. If you don't provide a specific version number, the client defaults to latest. For example, you can pull a specific version of ubuntu image

docker pull ubuntu:18.04

To get a new Docker image you can either get it from a registry (such as the Docker Hub) or create your own. There are tens of thousands of images available on [Docker Hub](https://hub.docker.com/explore/). You can also search for images directly from the command line using docker search.

An important distinction to be aware of when it comes to images is the difference between base and child images.

* **Base images** are images that have no parent image, usually images with an OS like ubuntu, busybox or debian.
* **Child images** are images that build on base images and add additional functionality.

Then there are official and user images, which can be both base and child images.

* **Official images** are images that are officially maintained and supported by the folks at Docker. These are typically one word long. In the list of images above, the python, ubuntu, busybox and hello-world images are official images.
* **User images** are images created and shared by users like you and me. They build on base images and add additional functionality. Typically, these are formatted as user/image-name.

### Our First Image

**Repo**: https://github.com/AjaySingala/DockerNotes/tree/master/For%20Demos/Py3.x

Now that we have a better understanding of images, it's time to create our own. Our goal in this section will be to create an image that sandboxes a simple [Flask](http://flask.pocoo.org/) application. For the purposes of this workshop, I've already created a fun little [Flask app](https://github.com/prakhar1989/docker-curriculum/tree/master/flask-app) that displays a random cat .gif every time it is loaded - because you know, who doesn't like cats?

The next step now is to create an image with this web app. As mentioned above, all user images are based on a base image. Since our application is written in Python, the base image we're going to use will be [Python 3](https://hub.docker.com/_/python/).

#### Dockerfile

A [Dockerfile](https://docs.docker.com/engine/reference/builder/) is a simple text file that contains a list of commands that the Docker client calls while creating an image. It's a simple way to automate the image creation process. The best part is that the [commands](https://docs.docker.com/engine/reference/builder/#from) you write in a Dockerfile are almost identical to their equivalent Linux commands. This means you don't really have to learn new syntax to create your own dockerfiles.

The application directory does contain a Dockerfile but since we're doing this for the first time, we'll create one from scratch. To start, create a new blank file in our favorite text-editor and save it in the **same** folder as the flask app by the name of Dockerfile.

We start with specifying our base image. Use the FROM keyword to do that -

FROM python:3.8

The next step usually is to write the commands of copying the files and installing the dependencies. First, we set a working directory and then copy all the files for our app.

# set a directory for the app

WORKDIR /usr/src/app

# copy all the files to the container

COPY . .

Now, that we have the files, we can install the dependencies.

# install dependencies

RUN pip install --no-cache-dir -r requirements.txt

The next thing we need to specify is the port number that needs to be exposed. Since our flask app is running on port 5000, that's what we'll indicate.

EXPOSE 5000

The last step is to write the command for running the application, which is simply - python ./app.py. We use the [CMD](https://docs.docker.com/engine/reference/builder/#cmd) command to do that -

CMD ["python", "./app.py"]

The primary purpose of CMD is to tell the container which command it should run when it is started. With that, our Dockerfile is now ready. This is how it looks -

FROM python:3.8

# set a directory for the app

WORKDIR /usr/src/app

# copy all the files to the container

COPY . .

# install dependencies

RUN pip install --no-cache-dir -r requirements.txt

# define the port number the container should expose

EXPOSE 5000

# run the command

CMD ["python", "./app.py"]

**Walk-thru app.py and requirements.txt**

Now that we have our Dockerfile, we can build our image. The docker build command does the heavy-lifting of creating a Docker image from a Dockerfile.

docker build -t <yourusername>/catnip .

docker build -t asingala/catnip .

If you don't have the python:3.8 image, the client will first pull the image and then create your image. Hence, your output from running the command will look different from mine. If everything went well, your image should be ready! Run docker images and see if your image shows.

The last step in this section is to run the image and see if it actually works (replacing my username with yours).

docker run -p 8888:5000 --name catnip <username>/catnip

docker run -p 8888:5000 --name catnip asingala/catnip

The command we just ran used port 5000 for the server inside the container and exposed this externally on port 8888. Head over to the URL with port 8888, where your app should be live.

Now we can see the ports by running the docker port [CONTAINER] command in a new window:

docker port catnip

To stop the container, determine its ID by running docker ps -a and then running either docker stop <container\_id> or docker container stop <container\_id>.

docker ps -a

docker stop <container\_id>

docker stop <container\_name>

OR

docker container stop <container\_id>

docker container stop <container\_name>

docker container stop <container\_id>

docker container stop catnip

To delete the container:

docker rm <container\_id>

docker rm <container\_name>

docker container rm <container\_id>

docker container rm <container\_name>

docker rm <container\_id>

docker rm catnip

Congratulations! You have successfully created your first docker image.

### Another Example: Py Flask + Redis

<https://www.youtube.com/watch?v=kTp5xUtcalw&t=5640s>

<https://github.com/K8sAcademy/Fundamentals-HandsOn>

**Source**: For Demos\DockerCompose

* Run docker compose (*builds and runs the containers*. No need for docker compose build)

# The -d is for detached mode.

docker compose up -d

* Navigate to localhost:5000 in browser. Refresh a few times.
* List the containers.

docker compose ps

docker ps

* Look at the container logs.

docker compose logs -f web-fe

* Split windows to show browser and terminal side-by-side.
* Refresh browser a few times.
* Notice the logs updated with every refresh.
* CTRL+C in terminal to stop logs display.
* List current projects:
  + Docker uses the current folder name as the project name.

docker compose ls

* Let’s try to deploy a second version:

docker compose up -d

* Says the app is already running.
* Deploy a second version using a different project name.

# -p : project name.

docker compose -p test up -d

* Will fail because port 5000 is already allocated.
* Change the port in the docker-compose.yaml file:

ports:

- 5001:5000

* In browser, navigate to localhost:5001.
* Refresh a few times. Works.
* Switch to localhost:5000 and refresh. That works too!

docker compose ls

* Clean up

docker compose down

docker compose ls

docker compose -p test down

docker compose ls

docker compose ps

docker ps

## Containers

**Repo**: https://github.com/AjaySingala/DockerNotes/tree/master/For%20Demos/Py2.7

* Create Dockerfile, requirements.txt and app.py

ls

docker build –t friendlyhello .

docker image ls

docker run -p 4000:80 friendlyhello

**Note**: If you are using Docker Toolbox on Windows 7, use the Docker Machine IP instead of localhost. For example, http://192.168.99.100:4000/. To find the IP address, use the command docker-machine ip.

<http://localhost:4000>

* OR (*from another Git Bash window or Command Prompt*)

curl http://localhost:4000

### To stop the container on Windows 10

* CTRL+C
* docker container ls
* note the container id
* docker container stop <container NAME or ID>

Additional commands:

docker container kill <hash> # Force shutdown of the specified container

docker container stop <hash> # Stop specified container

docker container rm <hash> # Remove specified container from this machine

docker container rm $(docker container ls -a -q) # Remove all containers

docker container start <hash> # Start specified container

docker image ls -a # List all images on this machine

docker image rm <image id> # Remove specified image from this machine

docker image rm $(docker image ls -a -q) # Remove all images from this machine

### Now, run app in the background

docker run -d -p 4000:80 friendlyhello

Then stop the container as explained above over [here](#_To_stop_the).

## Execute commands on the container

docker exec <container id> <cmd>

docker exec <container id> ls

docker exec <container id> sh

# interactive mode

docker exec -i <container id> sh

# tty input. Works from cmd not bash

docker exec -i -t <container id> sh -c "echo Hello"

## Share your image

### Log on to docker

docker login

### Tag the image

syntax: docker tag image username/repository:tag

docker tag friendlyhello asingala/get-started:part2

docker image ls

### Publish the image

Syntax: docker push username/repository:tag

docker push asingala/get-started:part2

### Pull and run the image from the remote repository

docker run -p 4000:80 asingala/get-started:part2

### Delete the image and container locally and then Pull and run from docker hub

docker container ls (*note the id, if exists*)

docker container stop <ID>

docker image ls –a (*note the id, if exists*)

docker image rm <ID> -f (*force stop*)

docker run -p 4000:80 username/get-started:part2

## Services

Scale our application and enable load-balancing

In a distributed application, different pieces of the app are called “services.” For example, if you imagine a video sharing site, it probably includes a service for storing application data in a database, a service for video transcoding in the background after a user uploads something, a service for the front-end, and so on.

* Services are really just “containers in production”.
* A service only runs one image, but it codifies the way that image runs
  + what ports it should use,
  + how many replicas of the container should run so the service has the capacity it needs, and so on
* Scaling a service changes the number of container instances running that piece of software, assigning more computing resources to the service in the process

Luckily, it’s very easy to define, run, and scale services with the Docker platform -- just write a docker-compose.yml file.

A docker-compose.yml file is a YAML (*Yet Another Markup Language*) file that defines how Docker containers should behave in production.

* Create docker-compose.yml
  + Indentation is very very important
  + No tabs allowed in yml files
  + Be sure you have pushed the image you created in the earlier section [Share Your Image](#_Share_your_image) to a registry, and update this .yml by replacing username/repo:tag with your image details.

This docker-compose.yml file tells Docker to do the following:

* Pull the image we uploaded earlier from the registry
* Run 5 instances of that image as a service called web, limiting each one to use, at most, 10% of the CPU (across all cores), and 50MB of RAM
* Immediately restart containers if one fails
* Map port 4000 on the host to web’s port 80
* Instruct web’s containers to share port 80 via a load-balanced network called webnet (Internally, the containers themselves publish to web’s port 80 at an ephemeral port.)
* Define the webnet network with the default settings (which is a load-balanced overlay network)

### Run your new load-balanced app

**Source**: DockerNotes/For Demos/docker-compose.yml

docker swarm init

# run single service stack run ing 5 container instances of the deployed image on one host

docker stack deploy -c docker-compose.yml getstartedlab

# Get the service ID for the one service in our application

docker service ls

# Look for output for the web service, prepended with your app name. If you named it the same as shown in this example, the name is getstartedlab\_web. The service ID is listed as well, along with the number of replicas, image name, and exposed ports.

# A single container running in a service is called a **task**. Tasks are given unique IDs that numerically increment, up to the number of replicas you defined in docker-compose.yml. List the tasks for your service:

docker service ps getstartedlab\_web

# Tasks also show up if you just list all the containers on your system, though that is not filtered by service:

docker container ls -a

docker container ls -q

# You can run curl -4 http://localhost:4000 several times in a row, or go to that URL in your browser and hit refresh a few times

Either way, the container ID changes, demonstrating the load-balancing; with each request, one of the 5 tasks is chosen, in a round-robin fashion, to respond. The container IDs match your output from the previous command (docker container ls -q).

docker inspect <task or container> # Inspect task or container

docker container ls –q (*list* *ids of all services*)

curl -4 <http://localhost:4000> (*-4 is to resolve name to IPv4 address*)

### Scale the app

* Change value of replicas in the yml file
* You can scale the app by changing the replicas value in docker-compose.yml, saving the change, and re-running the docker stack deploy command
* Docker performs an in-place update, no need to tear the stack down first or kill any containers
* Now, re-run docker container ls -q to see the deployed instances reconfigured. If you scaled up the replicas, more tasks, and hence, more containers, are started
* Re-run:

docker stack deploy -c docker-compose.yml getstartedlab

docker service ls

docker container ls -q

### Take down the app and the swarm

docker stack rm getstartedlab #Take down the app

docker swarm leave –f # (force) Take down the swarm

## Swarms

Deploy this application onto a cluster, running it on multiple machines. Multi-container, multi-machine applications are made possible by joining multiple machines into a “Dockerized” cluster called a **swarm**.

### Understanding Swarm clusters

A swarm is a group of machines that are running Docker and joined into a cluster. After that has happened, you continue to run the Docker commands you’re used to, but now they are 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**.

Swarm managers can use several strategies to run containers, such as “emptiest node” -- which fills the least utilized machines with containers. Or “global”, which ensures that each machine gets exactly one instance of the specified container. You instruct the swarm manager to use these strategies in the Compose file, just like the one you have already been using.

Swarm managers are the only machines in a swarm that can execute your commands, or authorize other machines to join the swarm as **workers**. Workers are just there to provide capacity and do not have the authority to tell any other machine what it can and cannot do.

Up until now, you have been using Docker in a single-host mode on your local machine. But Docker also can be switched into **swarm mode**, and that’s what enables the use of swarms. Enabling swarm mode instantly makes the current machine a swarm manager. From then on, Docker runs the commands you execute on the swarm you’re managing, rather than just on the current machine.

### Setup your swarm

#### Create a cluster

**VMS ON YOUR LOCAL MACHINE (WINDOWS 10)**

First, quickly create a virtual switch for your virtual machines (VMs) to share, so they can connect to each other.

1. Launch Hyper-V Manager
2. Click **Virtual Switch Manager** in the right-hand menu
3. Click **Create Virtual Switch** of type **External**
4. Give it the name myswitch, and check the box to share your host machine’s active network adapter

Now, create a couple of VMs using our node management tool:

**NOTE**:

docker-machine may give an error: Error with pre-create check: "Hyper-V PowerShell Module is not available"

If it does,

* go to <https://github.com/docker/machine/releases/tag/v0.13.0>
* download [**docker-machine-Windows-x86\_64.exe**](https://github.com/docker/machine/releases/download/v0.13.0/docker-machine-Windows-x86_64.exe)
* In Windows Explorer, go to C:\Program Files\Docker\Docker\resources\bin
* Rename docker-machine.exe to docker-machine.exe.org (*or anything that you want*)
* Copy the downloaded exe to this folder and rename it to docker-machine.exe
* Then run the docker-machine command again

docker-machine create -d hyperv --hyperv-virtual-switch "myswitch" myvm1

docker-machine create -d hyperv --hyperv-virtual-switch "myswitch" myvm2

**LIST THE VMS AND GET THEIR IP ADDRESSES**

docker-machine ls

**INITIALIZE THE SWARM AND ADD NODES**

docker-machine ssh myvm1 "docker swarm init --advertise-addr <myvm1 ip>"

**NOTE**:

Always run docker swarm init and docker swarm join with port 2377 (the swarm management port), or no port at all and let it take the default.

The machine IP addresses returned by docker-machine ls include port 2376, which is the Docker daemon port. Do not use this port or [you may experience errors](https://forums.docker.com/t/docker-swarm-join-with-virtualbox-connection-error-13-bad-certificate/31392/2).

As you can see, the response to docker swarm init contains a pre-configured docker swarm join command for you to run on any nodes you want to add. Copy this command, and send it to myvm2 via docker-machine ssh to have myvm2 join your new swarm as a worker:

docker-machine ssh myvm2 "docker swarm join \

--token <token> \

<ip>:2377"

Run docker node ls on the manager to view the nodes in this swarm

docker-machine ssh myvm1 "docker node ls"

### Deploy your app on the swarm cluster

#### Configure a docker-machine shell to the swarm manager

Run docker-machine env myvm1 to get the command to configure your shell to talk to myvm1.

docker-machine env myvm1

Gives result similar to this:

$Env:DOCKER\_TLS\_VERIFY = "1"

$Env:DOCKER\_HOST = "tcp://192.168.203.207:2376"

$Env:DOCKER\_CERT\_PATH = "C:\Users\sam\.docker\machine\machines\myvm1"

$Env:DOCKER\_MACHINE\_NAME = "myvm1"

$Env:COMPOSE\_CONVERT\_WINDOWS\_PATHS = "true"

# Run this command to configure your shell:

# & "C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe" env myvm1 | Invoke-Expression

Run the given command to configure your shell to talk to myvm1.

& "C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe" env myvm1 | Invoke-Expression

Run docker-machine ls to verify that myvm1 is the active machine as indicated by the asterisk next to it.

docker-machine ls

#### Deploy the app on the swarm manager

Just like before, run the following command to deploy the app on myvm1.

docker stack deploy -c docker-compose.yml getstartedlab

Now you can use the same [docker commands you used in the “Services part](#_Services). Only this time notice that the services (and associated containers) have been distributed between both myvm1 and myvm2.

docker stack ps getstartedlab

## Accessing your cluster

You can access your app from the IP address of **either** myvm1 or myvm2.

The network you created is shared between them and load-balancing. Run docker-machine ls to get your VMs’ IP addresses and visit either of them on a browser, hitting refresh (or just curl them).

In browser: <http://192.168.99.101> replace with relevant ip addresses for myvm1 and myvm2 (*use port no. 4000 or whichever has been set in the yml file*)

curl -4 http://192.168.99.101

## Cleanup and reboot

### Stacks and swarms

You can tear down the stack with docker stack rm. For example:

docker stack rm getstartedlab

At some point later, you can remove this swarm if you want to with docker-machine ssh myvm2 "docker swarm leave" on the worker and docker-machine ssh myvm1 "docker swarm leave --force" on the manager

### Unsetting docker-machine shell variable settings

You can unset the docker-machine environment variables in your current shell with the given command.

On **Mac or Linux** the command is:

eval $(docker-machine env -u)

On **Windows** the command is:

& "C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe" env -u | Invoke-Expression

This disconnects the shell from docker-machine created virtual machines, and allows you to continue working in the same shell, now using native docker commands (for example, on Docker for Mac or Docker for Windows).

## Restarting Docker machines

If you shut down your local host, Docker machines stops running. You can check the status of machines by running docker-machine ls.

$ docker-machine ls

NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS

myvm1 - virtualbox Stopped Unknown

myvm2 - virtualbox Stopped Unknown

To restart a machine that’s stopped, run:

docker-machine start <machine-name>

For example:

$ docker-machine start myvm1

$ docker-machine start myvm2

## Stacks

### Add a new service and redeploy

It’s easy to add services to our docker-compose.yml file. First, let’s add a free visualizer service that lets us look at how our swarm is scheduling containers.

1. Open up docker-compose.yml in an editor and replace its contents with the following. Be sure to replace username/repo:tag with your image details.

version: "3"

services:

web:

# replace username/repo:tag with your name and image details

image: username/repo:tag

deploy:

replicas: 5

restart\_policy:

condition: on-failure

resources:

limits:

cpus: "0.1"

memory: 50M

ports:

- "80:80"

networks:

- webnet

visualizer:

image: dockersamples/visualizer:stable

ports:

- "8080:8080"

volumes:

- "/var/run/docker.sock:/var/run/docker.sock"

deploy:

placement:

constraints: [node.role == manager]

networks:

- webnet

networks:

webnet:

1. Make sure your shell is configured to talk to myvm1 (full examples are [here](https://docs.docker.com/get-started/part4/#configure-a-docker-machine-shell-to-the-swarm-manager)).

* Run docker-machine ls to list machines and make sure you are connected to myvm1, as indicated by an asterisk next to it.
* If needed, re-run docker-machine env myvm1, then run the given command to configure the shell.

On **Mac or Linux** the command is:

eval $(docker-machine env myvm1)

On **Windows** the command is:

& "C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe" env myvm1 | Invoke-Expression

1. Re-run the docker stack deploy command on the manager, and whatever services need updating are updated:

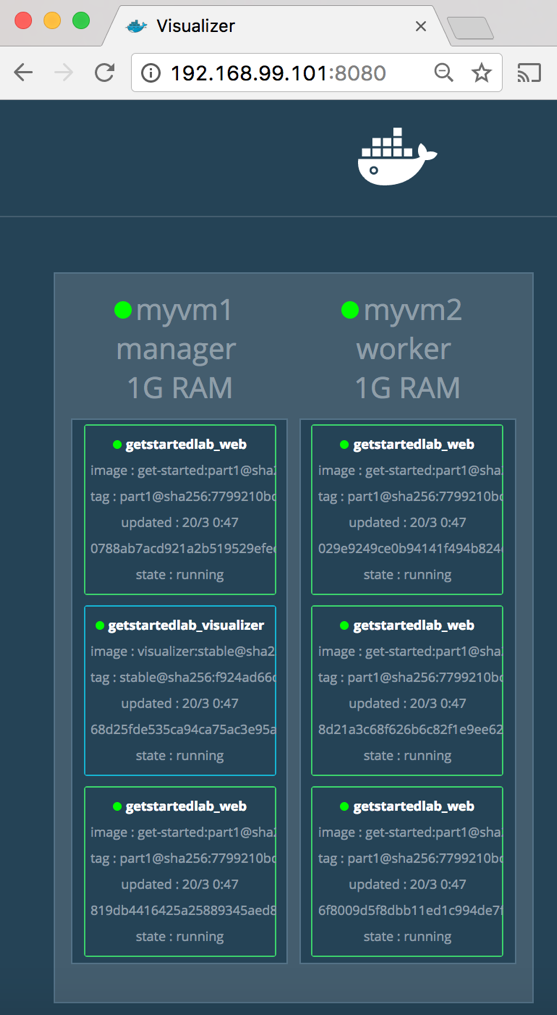
$ docker stack deploy -c docker-compose.yml getstartedlab

Updating service getstartedlab\_web (id: angi1bf5e4to03qu9f93trnxm)

Creating service getstartedlab\_visualizer (id: l9mnwkeq2jiononb5ihz9u7a4)

1. Take a look at the visualizer (*takes some time to refresh*)

You saw in the Compose file that visualizer runs on port 8080. Get the IP address of one of your nodes by running docker-machine ls. Go to either IP address at port 8080 and you can see the visualizer running:



The single copy of visualizer is running on the manager as you expect, and the 5 instances of web are spread out across the swarm. You can corroborate this visualization by running docker stack ps <stack>:

docker stack ps getstartedlab

The visualizer is a standalone service that can run in any app that includes it in the stack. It doesn’t depend on anything else. Now let’s create a service that does have a dependency: the Redis service that provides a visitor counter.

### Persist the data

Let’s go through the same workflow once more to add a Redis database for storing app data.

1. Save this new docker-compose.yml file, which finally adds a Redis service. Be sure to replace username/repo:tag with your image details.

version: "3"

services:

web:

# replace username/repo:tag with your name and image details

image: username/repo:tag

deploy:

replicas: 5

restart\_policy:

condition: on-failure

resources:

limits:

cpus: "0.1"

memory: 50M

ports:

- "80:80"

networks:

- webnet

visualizer:

image: dockersamples/visualizer:stable

ports:

- "8080:8080"

volumes:

- "/var/run/docker.sock:/var/run/docker.sock"

deploy:

placement:

constraints: [node.role == manager]

networks:

- webnet

redis:

image: redis

ports:

- "6379:6379"

volumes:

- "/home/docker/data:/data"

deploy:

placement:

constraints: [node.role == manager]

command: redis-server --appendonly yes

networks:

- webnet

networks:

webnet:

Redis has an official image in the Docker library and has been granted the short image name of just redis, so no username/repo notation here. The Redis port, 6379, has been pre-configured by Redis to be exposed from the container to the host, and here in our Compose file we expose it from the host to the world, so you can actually enter the IP for any of your nodes into Redis Desktop Manager and manage this Redis instance, if you so choose.

Most importantly, there are a couple of things in the redis specification that make data persist between deployments of this stack:

* + redis always runs on the manager, so it’s always using the same filesystem.
  + redis accesses an arbitrary directory in the host’s file system as /data inside the container, which is where Redis stores data.

Together, this is creating a “source of truth” in your host’s physical filesystem for the Redis data. Without this, Redis would store its data in /data inside the container’s filesystem, which would get wiped out if that container were ever redeployed.

This source of truth has two components:

* + The placement constraint you put on the Redis service, ensuring that it always uses the same host.
  + The volume you created that lets the container access ./data (on the host) as /data(inside the Redis container). While containers come and go, the files stored on ./data on the specified host persists, enabling continuity.

You are ready to deploy your new Redis-using stack.

1. Create a ./data directory on the manager:

docker-machine ssh myvm1 "mkdir ./data"

1. Make sure your shell is configured to talk to myvm1 (full examples are [here](https://docs.docker.com/get-started/part4/#configure-a-docker-machine-shell-to-the-swarm-manager)).
   * Run docker-machine ls to list machines and make sure you are connected to myvm1, as indicated by an asterisk next it.
   * If needed, re-run docker-machine env myvm1, then run the given command to configure the shell.

On **Mac or Linux** the command is:

eval $(docker-machine env myvm1)

On **Windows** the command is:

& "C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe" env myvm1 | Invoke-Expression

1. Run docker stack deploy one more time.

$ docker stack deploy -c docker-compose.yml getstartedlab

1. Run docker service ls to verify that the three services are running as expected.

$ docker service ls

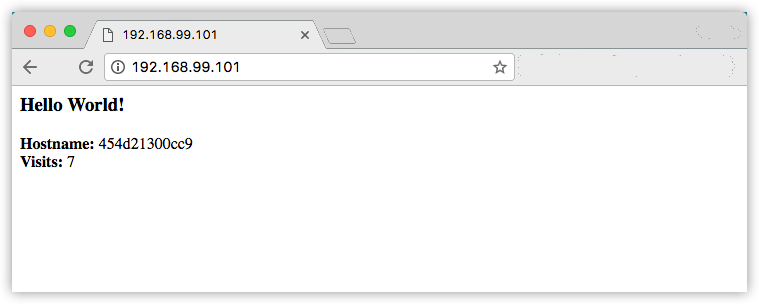
ID NAME MODE REPLICAS IMAGE PORTS

x7uij6xb4foj getstartedlab\_redis replicated 1/1 redis:latest \*:6379->6379/tcp

n5rvhm52ykq7 getstartedlab\_visualizer replicated 1/1 dockersamples/visualizer:stable \*:8080->8080/tcp

mifd433bti1d getstartedlab\_web replicated 5/5 gordon/getstarted:latest \*:80->80/tcp

1. Check the web page at one of your nodes, such as http://192.168.99.101, and take a look at the results of the visitor counter, which is now live and storing information on Redis. (*takes some time to refresh*)



Also, check the visualizer at port 8080 on either node’s IP address, and notice see the redis service running along with the web and visualizer services.



# Using Multiple Containers with Docker Compose

## MySQL and .NET

<https://www.freecodecamp.org/news/run-multiple-containers-with-docker-compose/>

**Source**: C:\Users\AjayJayantilalSingal\DockerNotes\For Demos\MySQL\_NodeJS\_dotNET

* Walk thru the .NET app code, connection string etc.
* Walk thru the .NET Core app’s Dockerfile.
* Walk thru the DB scripts in the sql-scripts folder.
* Walk thru the docker-compose.yml file.
* Run:

docker volume ls

# Remove the volume with the “\_datafiles” suffix.

docker volume rm <volume\_name>

docker compose up –build

* In browser, navigate to <http://localhost:8080/api/job>. Empty.
* Create records using these scripts:

INSERT INTO `super-app`.`User` (`Id`,`FirstName`)

VALUES (1,'John');

INSERT INTO `super-app`.`User` (`Id`,`FirstName`)

VALUES(1,'John');

* Refresh <http://localhost:8080/api/job>.

## PostgreSQL and .NET

<https://docs.docker.com/language/dotnet/develop/>

**Source**: \For Demos\PostgreSQL\_dotNET

* Show folder structure.
* Walk thru dockerfile, docker-compose.yml, .dockerignore and .env.
* Walk thru Models/Student.cs, Data/SchoolContext.cs, Pages/Index.cshtml, Pages/Index.cshtml.cs and appsettings.json files.
  + Have to run this initially:

dotnet add package Npgsql.EntityFrameworkCore.PostgreSQL

* Create our data volume:

docker volume create postgres-data

* Now we’ll create a network that our application and database will use to talk to each other. The network is called a user-defined bridge network and gives us a nice DNS lookup service which we can use when creating our connection string.

docker network create postgres-net

* Now we can run PostgreSQL in a container and attach to the volume and network we created above. Docker pulls the image from Hub and runs it for you locally. In the following command, option -v is for starting the container with the volume.

docker run --rm -d -v postgres-data:/var/lib/postgresql/data \

--network postgres-net \

--name db \

-e POSTGRES\_USER=postgres \

-e POSTGRES\_PASSWORD=example \

postgres

* Now, let’s make sure that our PostgreSQL database is running and that we can connect to it. Connect to the running PostgreSQL database inside the container using the following command:

docker exec -ti db psql -U postgres

psql (14.5 (Debian 14.5-1.pgdg110+1))

Type "help" for help.

postgres=#

* Press CTRL-D to exit the interactive terminal.
* Now we can rebuild our image. Open a terminal, change directory to the dotnet-docker directory and run the following command:

docker build --tag dotnet-docker .

* List your running containers.

docker ps

* Inspect the image column and stop any container that is using the dotnet-docker image.

docker stop dotnet-app

* Now, let’s run our container on the same network as the database. This allows us to access the database by its container name.

docker run \

--rm -d \

--network postgres-net \

--name dotnet-app \

-p 5000:80 \

dotnet-docker

* Let’s test that the application works and is connecting to the database. Using a web browser, access http://localhost:5000.
* You now have an application accessing the database, but the database contains no entries. Let’s connect Adminer to manage our database and create a database entry.

docker run \

--rm -d \

--network postgres-net \

--name db-admin \

-p 8080:8080 \

adminer

* Using a web browser, access http://localhost:8080.
* The Adminer login page appears.
* Specify the following in the login page and then click **Login**:
  + System: PostgreSQL
  + Server: db
  + Username: postgres
  + Password: example
  + Database: my\_db
* The Schema: public page appears.
* In Tables and views, click Students. The Table: Students page appears.
* Click New item. The Insert: Students page appears.
* Specify a LastName, FirstMidName, and EnrollmentDate. Click Save.
* Verify that the student name appears in the application. Use a web browser to access http://localhost:5000.
* List and then stop the application, database, and Adminer containers.

docker ps

docker stop db-admin dotnet-app db

### Better productivity with Docker Compose

In this section, we’ll create a Compose file to start our dotnet-docker app, Adminer, and the PostgreSQL database using a single command.

Now, to start our application and to confirm that it is running properly, run the following command:

docker-compose up --build

We pass the --build flag so Docker will compile our image and then start the containers.

Now let’s test our application. Using a web browser, access http://localhost:5000 to view the page.

To stop the containers started by Docker Compose, press Ctrl+C in the terminal where we ran docker-compose up. To remove those containers after they have been stopped, run docker-compose down.

### Detached mode

You can run containers started by the docker-compose command in detached mode, just as you would with the docker command, by using the -d flag.

To start the stack, defined by the Compose file in detached mode, run:

docker-compose up --build -d

Then, you can use docker-compose stop to stop the containers and docker-compose down to remove them.

In the docker-compose.yml file, POSTGRES\_PASSWORD=${POSTGRES\_PASSWORD:?database password not set} means that if the environment variable POSTGRES\_PASSWORD is not set on the host, Docker Compose will display an error. This is OK, because we don’t want to hard-code default values for the password. We set the password value in the .env file, which is local to our machine. It is always a good idea to add .env to .gitignore to prevent the secrets being checked into the version control.

Build and run your application to confirm the changes are applied properly.

docker-compose up --build -d

Now let’s test our application. Using a web browser, access http://localhost:5000 to view the page.

Add a Student record and refresh the page to see the name show up.

# Docker Multi Stage Build

<https://earthly.dev/blog/docker-multistage/>

<https://github.com/LukeMwila/builder-pattern-example>

**Source**: For Demos\MultiStageBuild

## Before multi-stage builds

One of the most challenging things about building images is keeping the image size down. Each RUN, COPY, and ADD instruction in the Dockerfile adds a layer to the image, and you need to remember to clean up any artifacts you don’t need before moving on to the next layer. To write a really efficient Dockerfile, you have traditionally needed to employ shell tricks and other logic to keep the layers as small as possible and to ensure that each layer has the artifacts it needs from the previous layer and nothing else.

One way of reducing the size of your Docker images is through the use of what is informally known as the [builder pattern](https://en.wikipedia.org/wiki/Builder_pattern). The builder pattern uses two Docker images to create a base image for building assets and the second to run it. This pattern was previously implemented through the use of multiple Dockerfiles.

It was actually very common to have one Dockerfile to use for development (which contained everything needed to build your application), and a slimmed-down one to use for production, which only contained your application and exactly what was needed to run it. This has been referred to as the “builder pattern”. Maintaining multiple Dockerfiles is not ideal.

* Walk thru Dockerfile.build
* Walk thru Dockerfile.main
* Walk thru Build.sh

While using the builder pattern does give you the desired outcome, it presents additional challenges. This process introduces the management overhead that comes with maintaining multiple Dockerfiles—not to mention the cumbersome procedure of running through several Docker CLI commands, even if this can be streamlined by a shell script.

## How to Use Docker Multistage Builds

Now that you get the underlying concept, turn your attention to how this translates to the modern implementation of the builder pattern. What the former approach accomplishes with multiple Dockerfiles, the multistage feature does in one. You can get the same results with your builds without the added complexity.

Multistage builds make use of one Dockerfile with multiple FROM instructions. Each of these FROM instructions is a new build stage that can COPY artifacts from the previous stages. By going and copying the build artifact from the build stage, you eliminate all the intermediate steps such as downloading of code, installing dependencies, and testing. All these steps create additional layers, and you want to eliminate them from the final image.

The build stage is named by appending AS name-of-build to the FROM instruction. The name of the build stage can be used in a subsequent FROM and COPY command by providing a convenient way to identify the source layer for files brought into the image build. The final image is produced from the last stage executed in the Dockerfile.

Try taking the example from the previous section that used more than one Dockerfile for the React application and replacing the solution with one file that uses a multistage build.

* Walk thru Dockerfile

This Dockerfile has two FROM commands, with each one constituting a distinct build stage. These distinct commands are numbered internally, stage 0 and stage 1 respectively. However, stage 0 is given a friendly alias of **build**. This stage builds the application and stores it in the directory specified by the WORKDIR command. The resultant image is over 420 MB in size.

The second stage starts by pulling the official Nginx image from Docker Hub. It then copies the updated virtual server configuration to replace the default Nginx configuration. Then the COPY –from command is used to copy only the production-related application code from the image built by the previous stage. The final image is approximately 127 MB.

# Docker Deployment to Azure

<https://learn.microsoft.com/en-us/azure/developer/javascript/tutorial/tutorial-vscode-docker-node/tutorial-vscode-docker-node-01?tabs=bash>

<https://soltisweb.com/blog/detail/2022-03-11-creatingafullyautomatedcicdpipeline>