Table of Contents

Docker Basics

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
```

Containers

- 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.

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 username/get-started:part2
```

`docker image ls`

Publish the image

Syntax: `docker push username/repository:tag`

```
docker push username/get-started:part2
```

Pull and run the image from the remote repository

```
docker run -p 4000:80 username/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 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

```
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 -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**
- 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.

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. 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:
```

2. Make sure your shell is configured to talk to `myvm1` (full examples are [here]).
   - 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:

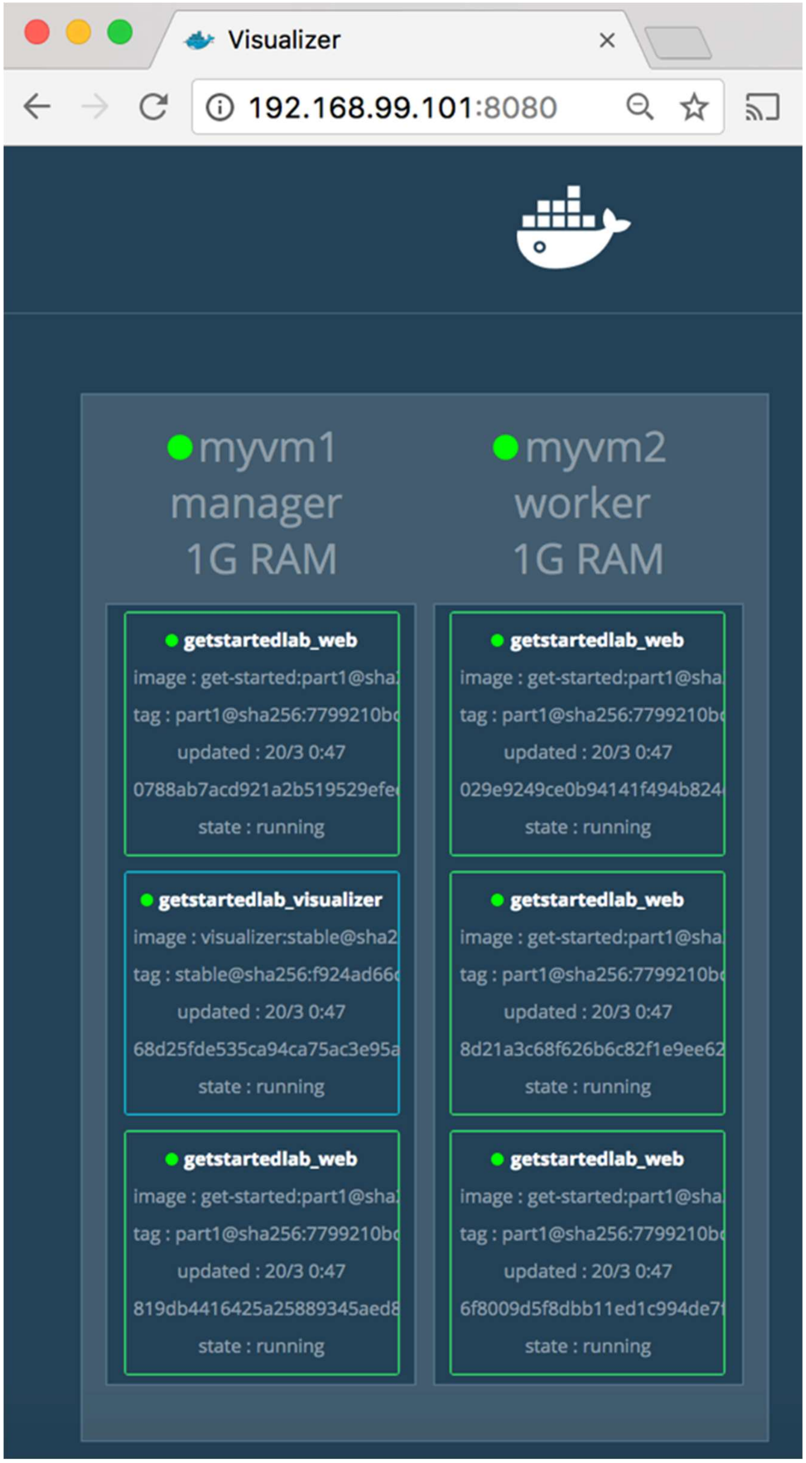
```
& "C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe" env
myvm1 | Invoke-Expression
```

3. 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)
```

4. 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.

2. Create a `./data` directory on the manager:

```
docker-machine ssh myvm1 "mkdir ./data"
```

3. Make sure your shell is configured to talk to `myvm1` (full examples are here).
   - 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
     ```

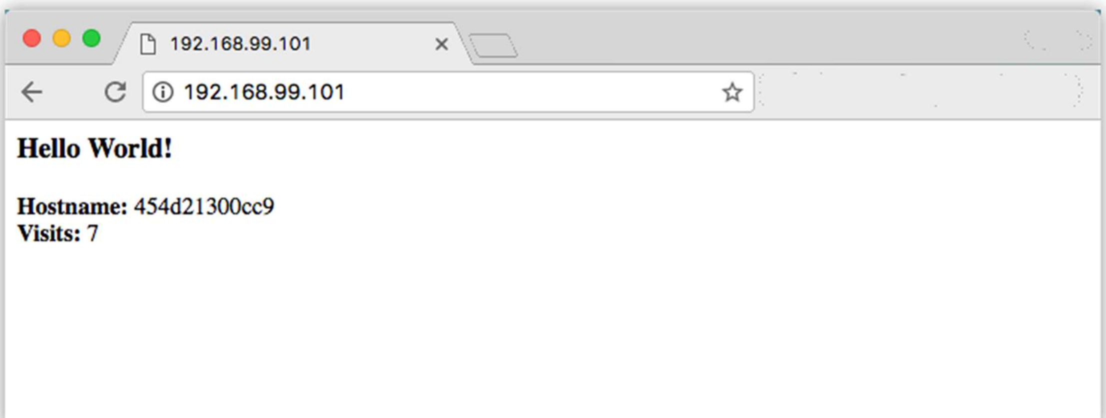
4. Run `docker stack deploy` one more time.

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

5. 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
```

6. 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.


Visualizer
192.168.99.100:8080
myvm1
manager
1G RAM
myvm2
worker
1G RAM
getstartedlab_visualizer
image : visualizer:stable@sha256:f92
tag : stable@sha256:f924ad66c8e94
updated : 2/6 16:3
87ffd7636236cf7d4465e823711c692
state : running
getstartedlab_web
image : get-started:part1@sha256:0
tag : part1@sha256:0601c866aab2a
updated : 2/6 16:3
dd4cf1a1c3f2dd6aaf1411205f96b03
state : running
getstartedlab_redis
image : redis:latest@sha256:548a75
tag : latest@sha256:548a75066f3f28
updated : 2/6 16:3
809cfb25a8fbc3426b67d6f8da69336
state : running
getstartedlab_web
image : get-started:part1@sha256:0
tag : part1@sha256:0601c866aab2a
updated : 2/6 16:3
dbc3451300ae1f2489212f4e79e622
state : running
getstartedlab_web
image : get-started:part1@sha256:0
tag : part1@sha256:0601c866aab2a
updated : 2/6 16:3
814539646a15503ebf30a29c0a35b8
state : running
getstartedlab_web
image : get-started:part1@sha256:0
tag : part1@sha256:0601c866aab2a
updated : 2/6 16:3
e1ba272633d6915263fdb2a04934b
state : running
getstartedlab_web
image : get-started:part1@sha256:0
tag : part1@sha256:0601c866aab2a
updated : 2/6 16:3
cc770a2a2a3bba2fb2a0961084f595
state : running

Hosting ASP.NET Core 2.0 Application In Docker

- At the command prompt, type

```
docker version
```

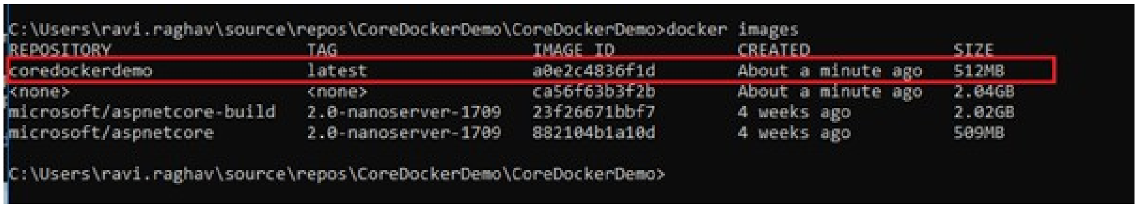
- VS 2017 -> C# -> .NET Core -> ASP.NET Core Web Application
- MVC Web App (CoreDockerDemo)
- Enable Docker Support
    - OS: Windows
    - No authentication
    - OK
- Docker files created:
    - Dockerfile
    - docker-compose
        - .dockerignore
        - docker-compose.yml
- Open Dockerfile

Dockerfile

```
FROM microsoft/aspnetcore:2.0-nanoserver-1709 AS base
WORKDIR /app
EXPOSE 80

FROM microsoft/aspnetcore-build:2.0-nanoserver-1709 AS build
WORKDIR /src
COPY CoreDockerDemo/CoreDockerDemo.csproj CoreDockerDemo/
RUN dotnet restore CoreDockerDemo/CoreDockerDemo.csproj
COPY . .
WORKDIR /src/CoreDockerDemo
RUN dotnet build CoreDockerDemo.csproj -c Release -o /app

FROM build AS publish
RUN dotnet publish CoreDockerDemo.csproj -c Release -o /app

FROM base AS final
WORKDIR /app
COPY --from=publish /app .
ENTRYPOINT ["dotnet", "CoreDockerDemo.dll"]
```

- Open command prompt
- Navigate to the folder where you created the solution. Then type:

```
docker-compose build

docker images
```

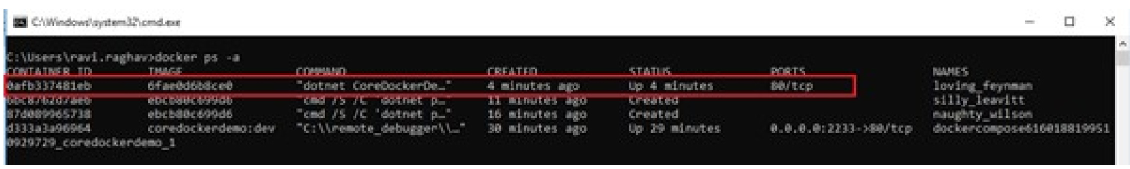


- Note the image id of your docker image
- App is now running. But this is just a template. We cannot connect to it. To run it, we have to put this image inside a container and then run the container

```
docker run <image id>
```

- Get the IP Address of our container. Open another command prompt and type:

```
docker ps -a
```



- Note the container id of the docker container
- To get the IP address of the container, type

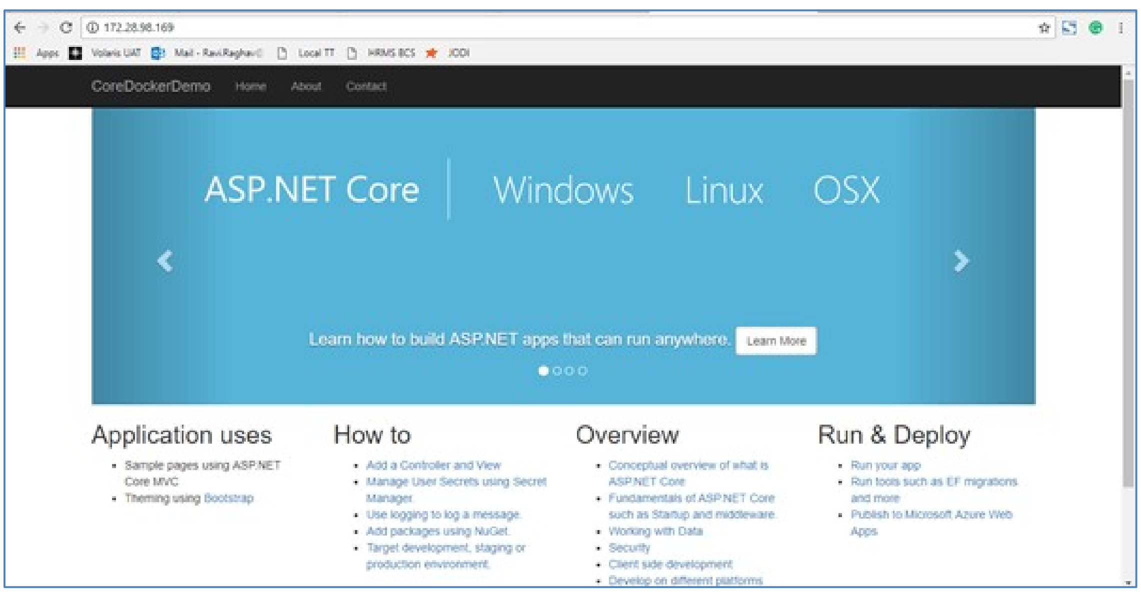
```
docker inspect <container id>
```

  - look for “Network Settings”, “Networks” and note the “IP Address” within that section

- Alternatively, type:

```
docker inspect -f "{{range
.NetworkSettings.Networks}}{{.IPAddress}}{{end}}"  your_container_id
```

- Open a browser and type IP address you noted



Make Changes and redeploy

- Make a change in VS
- Save (***do not build***)

- Switch to the command prompt and type:

```
docker-compose build

docker images
```

- Note the image id of your docker image
- Then, run the image:

```
docker run <image id>
```

- Get the IP Address of our container. Open another command prompt and type:

```
docker ps -a
```

- Note the container id of the docker container
- To get the IP address of the container, type

```
docker inspect <container id>
```

  - look for “Network Settings”, “Networks” and note the “IP Address” within that section
- Open a browser and type IP address you noted
- Your changes should be reflected

Azure DevOps, Docker and Angular

- Github: https://github.com/AjaySingala/ngDocker.git
- Pre-reqs:
  - VSCode Docker Extension installed
  - Docker for Windows installed and running
- Create an Azure Container Registry
- Admin user -> Enable
- Create -> Go to resource -> Access keys
- url: https://<reigstryname>.azurecr.io
- Note username and password
- Create a new folder for the Ng project
- ng new angularProject
- npm install
- ng serve
- navigate to http://localhost:4200 in a browser
- npm run build (*creates a "dist" folder*)
- Create dockerfile

FROM nginx:alpine	Start with a NGINX base image, lightweight Web server. Also acts as a reverse proxy and load balancer.
LABEL author="john smith"	Name of author
COPY ./dist /usr/share/nginx/html	Copy contents of the ./dist folder to the /usr/share/nginx/html folder of the image. When your Docker image is created, your files will be ready to serve out of that folder.
EXPOSE 80 443	Docker doesn't expose any ports from the container by default; you have to specify what you'd like to have exposed. Here I'm saying that I wish to expose the container's ports 80 and 443.
ENTRYPOINT ["nginx", "-g", "daemon off;"]	This final line of code is the ENTRYPOINT, which is a Docker concept. It sets the command and parameters that are run when the container is run. In other words, the container starts, and it starts nginx. Remember that the nginx:alpine image is so lightweight that, effectively, you should have a full-fledged Web server with the application running in a completely contained container that's mere kilobytes in size.

- CTRL+SHIFT+P -> Docker: Build -> tag as "angularProject:latest"
  - `docker build --rm -f "dockerfile" -t angularproject:latest .`
  - `docker run -d -p 4000:80 angularproject:latest`
  - Browse to http://localhost:4000
  - docker image ls
  - docker container ls –all
  - docker container stop <hash>

    - Browse to http://localhost:4000 -> ERROR
    - docker container start <hash>
    - Browse to http://localhost:4000 - WORKS
- Azure DevOps Build Pipeline
    - Project: ngDockerProject (start with empty)
    - Build Pipeline: ngDockerProject-CI
    - npm install @angular/cli --save-dev
    - npm run build
    - build an image
        - Service connection to ACR
        - Image name: $(Build.Repository.Name):latest
    - Push an image
        - Service connection to ACR
        - Image name: $(Build.Repository.Name):latest
    - Azure Portal -> Container Registry -> Repositories
        - Check new "latest" tag
    - Deploy:
        - Create a new Web App for Containers (Linux)
            - Name: <azure web app name>
            - RG: <azure registry name>
            - OS: Linux
            - Configure Container -> Azure Container Registry
            - Registry: <azure registry name>
            - Image: <username>/ngdocker
            - Tag: latest
            - Apply -> Create
            - Navigate to azure web app
        - On CI pipeline, enable CI
        - On latest successful CI build, select Release
            - CD Pipeline: ngDockerProject-CI - CD (start with empty)
            - On "Run on Agent", change Agent pool to "Hosted Ubuntu 1604"
            - Add task "Deploy Azure App Service"
            - Connection type: Azure Resource Manager
            - Azure Subs: ngDockerSvcConn
            - App Service Type: Web App for Containers (Linux)
            - App service name: <azure web app>
            - Registry: <azure registry>.axurecr.io
            - Image: username/ngdocker
            - Tag: $(Build.BuildId)
            - Save
            - Ensure CD is enabled
        - Make a change to code
            - src/app.component.html

  - <h1> I love Azure DevOps </h1>
- git add .
- git commit –m “h1 added”
- git push
- Should kick off build and then deployment
- Navigate to https://<azure web app>.azurewebsites.net/
- Changes reflected