Docker Swarm Tutorial | Code Along | Zero to Hero under 1 Hour

In this Docker Swarm tutorial we'll build the basic Swarm skills that you'll need in your project work. You'll learn to set up a Swarm cluster, Deploy applications, explore and manage your stack in the cluster, and we'll also go through typical maintenance activities that you'll need.

This is a code-along tutorial, so please open up your terminal and get ready. We'll use a two step approach:

1. We'll set up a cluster and deploy an application first. The goal here is to show you what Swarm is all about and what it's capable of. We'll set up a one machine cluster on your local machine first, and then we'll set up a cluster of two hosts.
2. Once you get the big picture in step 1, I'll explain all details behind. I think it's easier to learn and understand after you've seen the stuff in action.

If you want to jump to a specific step, please check out the table of contents above.

In order to get the most out of this article you should be familiar with basic Docker concepts, like containers, images, the Dockerfile and Docker Compose. If you need a refresher on these topics, please go to my previous tutorials first. Alternatively you can get my book for a complete guided experience.

This tutorial is available as a Youtube video, too.

Your first Swarm cluster

Swarm is Docker's built in container orchestrator solution, its main purpose is to manage containers in a computer cluster, i.e. a set of connected computers that work together.

Swarm comes built into the Docker Engine, you don't need to install anything to get started. (You need to have Docker installed, of course, but you don't need extra components.)

Let's go to terminal and turn your local computer into a one machine Swarm cluster.

```
$ docker swarm init

Swarm initialized: current node (dw78e2gl8jm13basl0nqmcivz) is now a manager.

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

    docker swarm join --token SWMTKN-1-0bwv7h6u9c49p49xc5hpapb1vsu9vdmcwtatuuh3ultmbq7j8m-9dmuana0kx039475l13cit1n2 192.168.65.3:2377

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

Computers in a Swarm cluster are called nodes. Nodes can play two roles in a Swarm:

- The role of **manager nodes** is to manage the cluster; you can execute Swarm management commands on manager nodes.

- The role of **worker nodes** is to run your containers that do the actual job (like running a web server).

A Swarm node acts as both worker and manager by default, the default behavior is to schedule worker containers on manager nodes, too.

The above command output tells you that this node is now a manager and it also gives you the command that you can use to add further machines to the Swarm.

Your local machine is the manager node in the cluster, and at the same time it is also the single worker node that you can deploy containers to. Let's deploy something cool!

Your first Swarm deployment

Let's create a Node Express application with the following files:

```
$ tree
.
├── Dockerfile
├── docker-compose.yml
├── package-lock.json
├── package.json
└── server.js
```

Let's create the `package.json` and `package-lock.json` files first with the use of a Docker container:

```
$ docker run --rm -v $(pwd):/home/node -w /home/node node:11.1.0-alpine npm init -y
$ docker run --rm -v $(pwd):/home/node -w /home/node node:11.1.0-alpine npm i -S express
```

The first command will initialize our `package.json` file with default values. The second command installs Express and saves Express as a dependency into `package.json`. The second command also generates `package-lock.json`.

Let's create `server.js` with the following code:

```
const express = require("express");
const os = require("os");

const app=express();

app.get("/", (req, res) => {
    res.send("Hello from Swarm " + os.hostname());
});

app.listen(3000, () => {
    console.log("Server is running on port 3000");
});
```

This piece of code will start up an Express server and will display a message that includes the identifier of the container where the server is running. We achieve this by adding `os.hostname()` to the response, `os.hostname()` in the Docker context will display the container ID rather than the host name. I added the container ID to the message, so that we can see where the responses are coming from as we scale our service to multiple replicas later.

Let's add the `Dockerfile` to the solution to copy our `package.json`, `package-lock.json` and `server.js` into a Node image, install Node dependencies and specify `npm start` as the default command. (Note, that the default value of `npm start` in the `package.json` file is `node server.js`, so we are just telling Docker to run our `server.js` file with Node.)

```
FROM node:11.1.0-alpine

WORKDIR /home/node

COPY . .

RUN npm install

CMD npm start
```

In order to define the build and runtime behavior of our project, we add the following Compose file:

```
version: '3'

services:
    web:
        build: .
        image: takacsmark/swarm-example:1.0
        ports:
            - 80:3000
        networks:
            - mynet

networks:
    mynet:
```

We specified one service in the Compose file for the Express server, mapped the port to port 80 and defined a default network for the stack.

Let's build our image.

```
$ docker-compose build
```

I you are familiar with Compose, you know that we can start this stack with the `docker-compose up` command, but we will not do this. We will start the stack in Swarm.

You might be asking yourself right now; **how are we going to start the stack in Swarm mode with a Compose file?**

Docker Swarm is designed in a very practical way; you can use the Compose file to start up your application stack in Swarm.

Let's do this and I'll explain what's happening. The command to start the stack is the following:

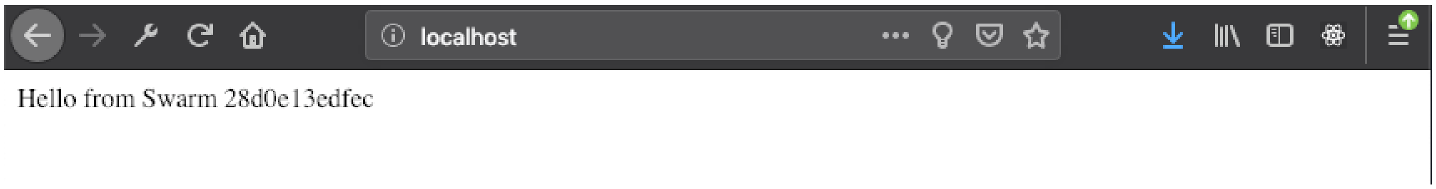
```
$ docker stack deploy nodeapp -c docker-compose.yml
Ignoring unsupported options: build

Creating network nodeapp_mynet
Creating service nodeapp_web
```

Where `docker stack` is the management command to manage your stack deployments in the Swarm. We used the `deploy` subcommand to deploy the stack and we named the stack `nodeapp`. You can use any name that you like. The rest of the command, `-c docker-compose.yml`, specifies the Compose file that describes the deployment.

Docker created the network that we defined in the Compose file and prefixed the network name with the name of the stack, hence the final name, `nodeapp_mynet`. Docker also created a service called `nodapp_web`. Please note that Docker Compose creates containers when you start up a multi-container application, while the Swarm created and entity that's called a service. A service is a new entity that we use to configure the behavior of our containers in the Swarm, we'll have a closer look at it in a minute.

Please also note the first line of the command output above that says `Ignoring unsupported options: build`. This line tells us that Swarm does not support the `build` option in the Compose file. I'll also tell you about the difference between Compose and Swarm in a minute.

Let's see the results of our work first, please visit http://localhost; you should see a message like this one: `Hello from Swarm 63daead91754`.



Hello from Swarm 28d0e13edfec

Explore the stack

We deployed a stack to the Swarm based on the description in the Compose file. We can see our deployed stack with a simple command:

```
$ docker stack ls
NAME                SERVICES            ORCHESTRATOR
nodeapp             1                   Swarm
```

The output says that we have one service running, which is in line with what we defined in the Compose file. Docker gives us more control over our stack, we can explore services in the stack in more details:

```
$ docker stack services nodeapp
ID                  NAME                MODE                REPLICAS            IMAGE
PORTS
```

```
og72l2onptng        nodeapp_web         replicated          1/1                 takacsmark/swarm-
example:1.0   *:80->3000/tcp
```
Now we have access to the service identifier, the service name, number of replicas, image and port information. We can go one level deeper and explore a lower level scheduling entity in the stack, called the task:

```
$ docker stack ps nodeapp
ID                  NAME                IMAGE                           NODE
DESIRED STATE       CURRENT STATE               ERROR               PORTS
nv0vu1micwh7        nodeapp_web.1       takacsmark/swarm-example:1.0   linuxkit-025000000001
Running             Running about an hour ago
```
As you can see, we have exactly one task running in our stack on the single node in our cluster right now.

Your first scaled service

Let’s scale our service to run multiple replicas.

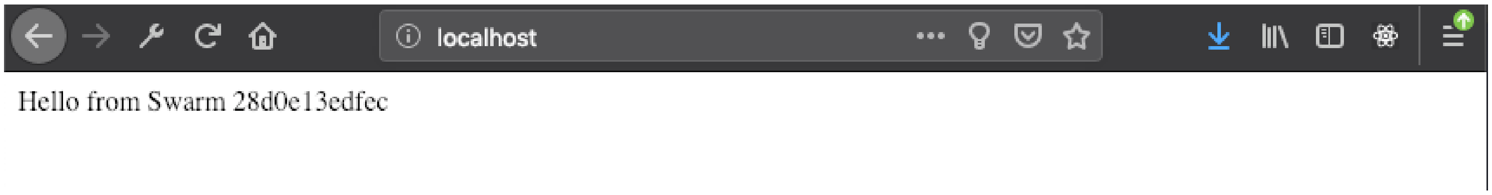
```
$ docker service scale nodeapp_web=4
nodeapp_web scaled to 4
overall progress: 4 out of 4 tasks
1/4: running   [==================================================>]
2/4: running   [==================================================>]
3/4: running   [==================================================>]
4/4: running   [==================================================>]
verify: Service converged
```
We used the `docker service scale` command to start more replicas of our service. Let’s check the service status:

```
$ docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE
PORTS
og72l2onptng        nodeapp_web         replicated          4/4                 takacsmark/swarm-
example:1.0   *:80->3000/tcp
```
The output says that we have 4/4 replicas running. Let’s check the individual tasks:

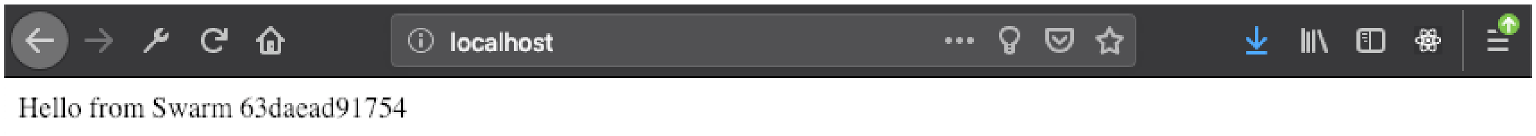
```
$ docker service ps nodeapp_web
ID                  NAME                IMAGE                           NODE
DESIRED STATE       CURRENT STATE               ERROR               PORTS
nv0vu1micwh7        nodeapp_web.1       takacsmark/swarm-example:1.0    linuxkit-025000000001
Running             Running about an hour ago
acd8h7clttsy        nodeapp_web.2       takacsmark/swarm-example:1.0    linuxkit-025000000001
Running             Running 2 minutes ago
ip2l6ldbxk6j        nodeapp_web.3       takacsmark/swarm-example:1.0    linuxkit-025000000001
Running             Running 2 minutes ago
rbwuk4v7wqu0        nodeapp_web.4       takacsmark/swarm-example:1.0    linuxkit-025000000001
Running             Running 2 minutes ago
```

Docker started four tasks automatically for our scaled service. Docker displays all tasks running on our single node in the cluster.

If you go to http://localhost and refresh the page a few times, you'll get alternating host names, because your request is served from different replicated containers. (Make sure to clear the browser cache during refresh to see the results, you can easily do this with CMD+SHIFT+R on Mac or CTRL+F5 on Windows and Linux, this will work with most browsers.)



Hello from Swarm 28d0e13edfec



Hello from Swarm 63daead91754

Set up a cluster with multiple hosts

Let's step one level higher and create a cluster with multiple machines. Let's remove our stack first and leave the Swarm on the local machine.

```
$ docker stack rm nodeapp
Removing service nodeapp_web
Removing network nodeapp_mynet
```

```
$ docker swarm leave --force
Node left the swarm.
```

We stay on the local machine for this exercise, I hope your computer has the power to run two virtual machines in parallel. If you don't have the power, you can do this exercise on Play with Docker.

Docker has a handy tool called Docker machine that lets us create virtual machines locally and also in the cloud. Docker machine comes bundled with Docker on Mac and Windows, and it needs to be installed on Linux. Please refer to the official documentation if you need to install Docker machine.

Let's start up two VMs locally. Please make sure you have Virtualbox installed before proceeding. Issue the below commands in terminal, please be patient it will take a while to create the machines.

```
$ docker-machine create --driver virtualbox myvm1
$ docker-machine create --driver virtualbox myvm2
```

Both machines should be up and running after creation. You can check their status with the `docker-machine ls` command.

Let's `ssh` into machine 1 and start our Swarm. This machine has multiple network interfaces, so we need to fine-tune to our command to advertise the right network address in the Swarm.

```
$ docker-machine ssh myvm1
   ( '>')
  /) TC (\   Core is distributed with ABSOLUTELY NO WARRANTY.
 (/-_--_-\)           www.tinycorelinux.net

docker@myvm1:~$ docker swarm init --advertise-addr eth1
Swarm initialized: current node (yttgwlrspn5ln3ho1qffiy4op) is now a manager.

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

    docker swarm join --token SWMTKN-1-678250sc60og59wbwf25z6xwiz5a1ufg2q6skgxh7ccu0lx7w5-
4dza21v2frlv5dbali96x8mw4 192.168.99.100:2377

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

Leave the machine with the `exit` command and `ssh` into machine 2 and join the Swarm on machine 1. We can do this by using the command from the above command output `docker swarm join --token SWMTKN-1-678250sc60og59wbwf25z6xwiz5a1ufg2q6skgxh7ccu0lx7w5-4dza21v2frlv5dbali96x8mw4 192.168.99.100:2377`. Please make sure that you copy the join command from your command output.

```
$ docker-machine ssh myvm2
   ( '>')
  /) TC (\   Core is distributed with ABSOLUTELY NO WARRANTY.
 (/-_--_-\)           www.tinycorelinux.net

docker@myvm2:~$ docker swarm join --token SWMTKN-1-
678250sc60og59wbwf25z6xwiz5a1ufg2q6skgxh7ccu0lx7w5-4dza21v2frlv5dbali96x8mw4 192.168.99.100:2377
This node joined a swarm as a worker.
```

Now we have two machines in the Swarm. In order to deploy our Node example application to the Swarm we need to make sure that our project’s Docker image is available to the virtual machines. Our workflow on a real project would be to automatically build and push our tested Docker images to a central repository. So we’ll do exactly this, we’ll push our Docker image to the Docker Hub into a public repository.

```
$ docker-compose push
```

Once the image is pushed, we want to find a way to use the Compose file that is stored on our host machine in the Docker machine VM. Docker machine has a feature to enable this. We just need to set our shell to talk to the Docker daemon in the VM. We use the `docker-machine env <vm_name>` command to configure the shell, and then grab the last line of the command output and execute it.

```
$ docker-machine env myvm1
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="/Users/takacsmark/.docker/machine/machines/myvm1"
export DOCKER_MACHINE_NAME="myvm1"
# Run this command to configure your shell:
# eval $(docker-machine env myvm1)
$ eval $(docker-machine env myvm1)
```

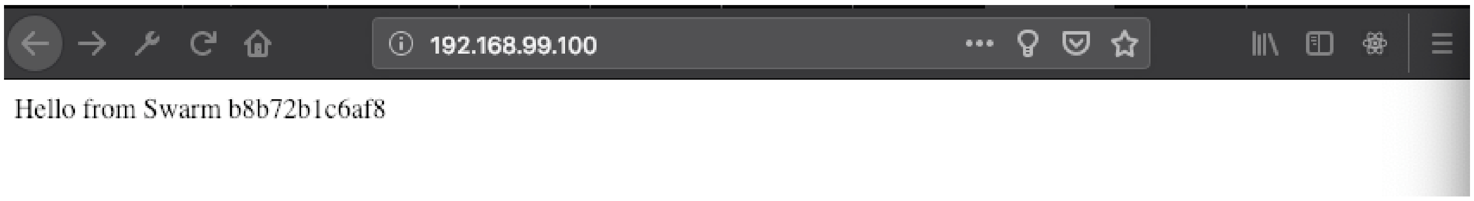
Now we can deploy our application the usual way.

```
$ docker stack deploy nodeapp -c docker-compose.yml
Ignoring unsupported options: build

Creating network nodeapp_mynet
Creating service nodeapp_web
```

Once the deployment is done, you can access your application at the address of the virtual machines. You can find out the address with a simple command.

```
$ docker-machine ls
NAME     ACTIVE    DRIVER       STATE     URL                         SWARM    DOCKER      ERRORS
myvm1    *         virtualbox   Running   tcp://192.168.99.100:2376            v18.09.1
myvm2    -         virtualbox   Running   tcp://192.168.99.101:2376            v18.09.1
```

You can use the IP address of any of the machines to access the application. Let's use the address of `myvm1` in the browser.



Hello from Swarm b8b72b1c6af8

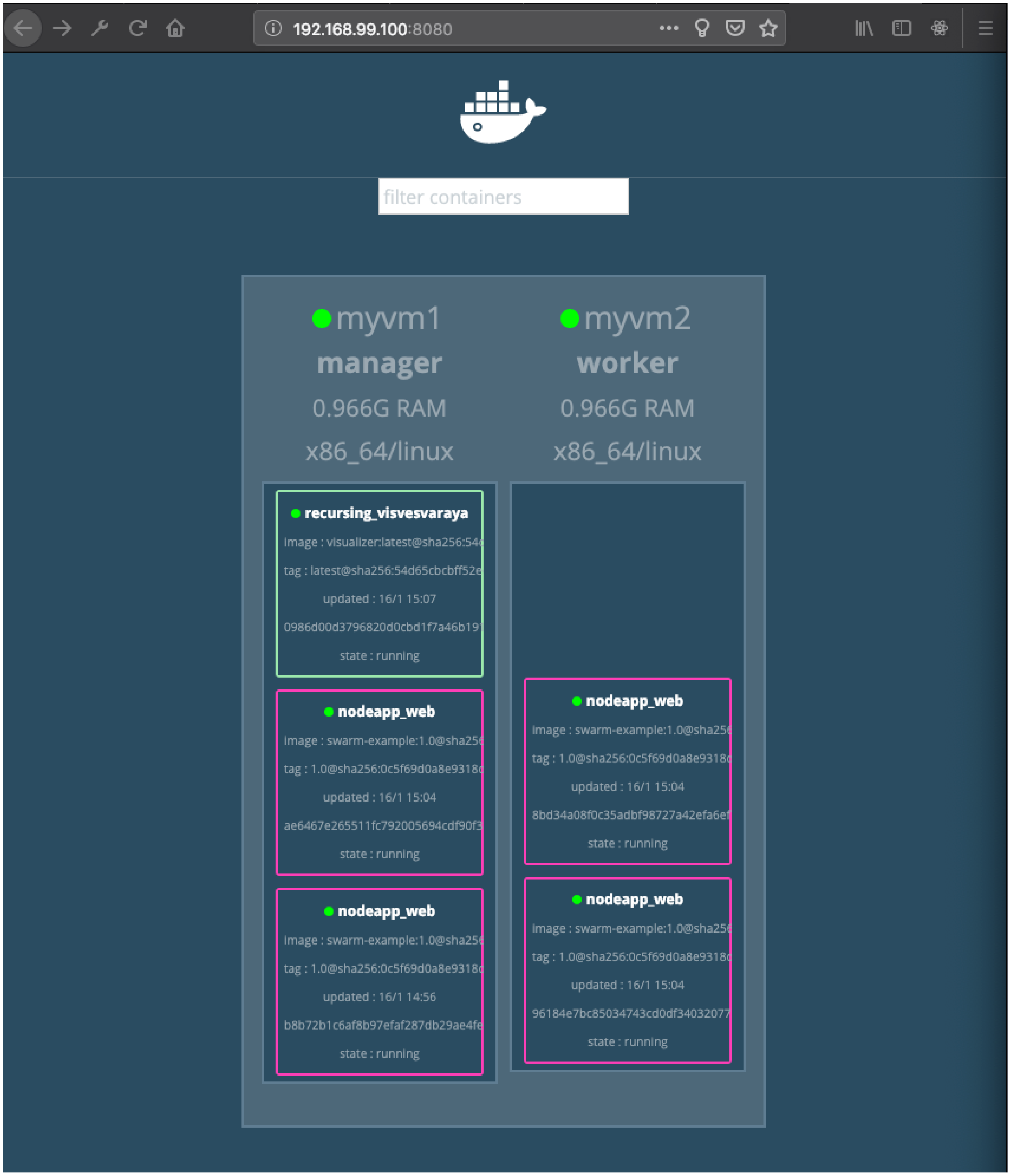
Why not load the cluster with a few replicas and check out the whole thing in a nice visual representation?

```
$ docker service scale nodeapp_web=4
```

We just scaled up our service, let's add another service to the Swarm to create a visualizer.

```
$ docker service create \
-p 8080:8080 \
--constraint=node.role==manager \
--mount=type=bind,src=/var/run/docker.sock,dst=/var/run/docker.sock \
dockersamples/visualizer
```

We started a container from the `dockersamples/visualizer` image, we gave it a constraint to run on a manager node and made it available on port 8080. Now if you visit port 8080 on any VM in the browser you'll get a visual webpage like the one below.



192.168.99.100:8080

filter containers

● myvm1
manager
0.966G RAM
x86_64/linux

● recursing_visvesvaraya
image : visualizer:latest@sha256:54d
tag : latest@sha256:54d65cbcbff52e
updated : 16/1 15:07
0986d00d3796820d0cbd1f7a46b19
state : running

● nodeapp_web
image : swarm-example:1.0@sha256
tag : 1.0@sha256:0c5f69d0a8e9318d
updated : 16/1 15:04
ae6467e265511fc792005694cdf90f3
state : running

● nodeapp_web
image : swarm-example:1.0@sha256
tag : 1.0@sha256:0c5f69d0a8e9318d
updated : 16/1 14:56
b8b72b1c6af8b97efaf287db29ae4fe
state : running

● myvm2
worker
0.966G RAM
x86_64/linux

● nodeapp_web
image : swarm-example:1.0@sha256
tag : 1.0@sha256:0c5f69d0a8e9318d
updated : 16/1 15:04
8bd34a08f0c35adbf98727a42efa6ef
state : running

● nodeapp_web
image : swarm-example:1.0@sha256
tag : 1.0@sha256:0c5f69d0a8e9318d
updated : 16/1 15:04
96184e7bc85034743cd0df34032077
state : running

You can clearly see on the figure that `myvm1` is the manger node in the cluster, `myvm2` is the worker and both nodes have containers scheduled on them. The visualizer is running on the manager node, because we defined a constraint to run on a node that has a manager node. (This constraint was needed by the way, because the visualizer can collect the Swarm info on a manager node only.)

Now that we have seen some of the basic functionality of Docker Swarm, let's move on and let's understand what's going on in the background.

What is Docker Swarm mode

Docker was designed with multi-host, horizontally scaled production operations in mind. This is a great benefit to your projects, because you can run multiple instances of your application containers and distribute them on multiple servers, so that you can cope with load peaks coming from user requests.

Docker has built-in features to manage your application across several computers, i.e. a computer cluster. This solution is called the Swarm mode.

Swarm comes integrated into the Docker Engine, so you don't need to install any new component to use Swarm, it's available out of the box.

What is the difference between Docker Compose and Swarm

We used Docker Compose to run applications in previous tutorials. **The key difference between Docker Compose and Swarm is that Compose is limited to a single host.**

You can use Compose to run multiple containers that are connected over a number of user defined networks, but this solution is always limited to a single host. If you need to run your application on a cluster of multiple hosts, you need Swarm.

As you have seen in the example, we can use the usual Compose file with both Compose and Swarm. You will find that certain options in the Compose file will only work with either Compose or Swarm. The unsupported options will be simply ignored by Docker commands.

The `build` option, for example, is supported by Compose only and you can use it to build your services' images as described in the Compose file. However if you run a stack with `build` settings in Swarm, you'll notice that you get the message that tells you that `build` is not supported by Swarm. Swarm will simply ignore the build configuration in the Compose file. We had this message in our example.

The `deploy` option, for example is only supported by Swarm. You can use the `deploy` setting to describe your deployment configuration in a Swarm. Compose does not support these settings and it will simply ignore this part of your Compose file.

The Compose file reference specifies exactly whether an option is limited to either Compose or Swarm. We use the same file reference in both cases and we always double check if the settings are applicable to Swarm or Compose only.

Services, tasks and containers

You saw that Docker created services when we deployed our stack with `docker stack deploy` in the example.

We introduced services with Docker Compose, and our definition was something like this: *Define your application's logical components in the Compose file as services. One service should be based on one Docker image and one service should be in charge of one specific aspect of your application.*

A good example of a service would be your Apache web server, a Redis database, a RabbitMQ queue, a Node app or other similar logical units of your application.

The notion of service in the Swarm follows the same concept. When in Swarm mode, Docker will provision and scale your application services automatically.

There is a dedicated command called `docker service` to manage services in the Swarm, we'll have a look at it in a minute.

Docker introduced a new term with Swarm, called the task. The task is a scheduling slot for your containers in the service. Each task invokes exactly one container.

The logical hierarchy of components is stack, service, task and container, so we'll look at these entities in this order from top to the bottom.

To see our stack(s) we use the `docker stack ls` command.

```
$ docker stack ls
NAME                SERVICES            ORCHESTRATOR
nodeapp             1                   Swarm
```

Our deployed stack is displayed in the output. It has one service (visualizer was added separately, that's why it's not displayed.) You can use multiple stacks in real projects to deploy various parts of your architecture. You may have a separate front-end, back-end and operations stack, for example.

We can list the services in our stack with `docker stack services nodeapp` or the services in the entire Swarm with `docker service ls`, let's try the latter.

```
$ docker service ls
ID                  NAME                     MODE                REPLICAS            IMAGE
PORTS
vtj81lstp1lw        nodeapp_web              replicated          4/4
takacsmark/swarm-example:1.0      *:80->3000/tcp
6rlou1x03dgf        recursing_visvesvaraya   replicated          1/1
dockersamples/visualizer:latest   *:8080->8080/tcp
```

This is the overview of all the services in our Swarm cluster. We are running four replicas of our Node application and one instance of visualizer.

Let's look at the tasks in the `nodeapp_web` service.

```
$ docker service ps nodeapp_web
4alkk6og8pae        nodeapp_web.1       takacsmark/swarm-example:1.0   myvm1               Running
Running about an hour ago
ixjb3ztdumfw        nodeapp_web.2       takacsmark/swarm-example:1.0   myvm2               Running
Running 44 minutes ago
fyl2c8gw7rqe        nodeapp_web.3       takacsmark/swarm-example:1.0   myvm1               Running
Running 44 minutes ago
1w8ou7h4g15f        nodeapp_web.4       takacsmark/swarm-example:1.0   myvm2               Running
Running 44 minutes ago
```

This output tells us that we have two tasks running on `myvm1` and another two tasks running on `myvm2`. Remember, a task is a logical slot where Docker schedules containers, so there is exactly one container running per each task, but a task is not something you can manage yourself. There is no `docker task` command to manage tasks.

If you want to change the set-up, you have to change the service configuration. If you need to run a command in a running container with `exec`, then you need to work with the container directly.

Manage your stack

We used the `docker stack deploy` command to deploy our stack. This command has a few more options to manage stacks. Let's have a look at the most important ones.

List stacks

Use the `docker stack ls` command to list stacks in your Swarm. Remember, you can use multiple stacks in your application architecture design.

List services in the stack

Use the `docker stack services` command to list the services in a stack.

```
$ docker stack services nodeapp
ID                  NAME                MODE                REPLICAS            IMAGE
PORTS
```

```
vtj81lstp1lw        nodeapp_web         replicated          4/4                 takacsmark/swarm-
example:1.0   *:80->3000/tcp
```

List tasks in a stack

Use the `docker stack ps` command to list tasks in a stack.

```
$ docker stack ps nodeapp
ID                  NAME                IMAGE                           NODE                DESIRED
STATE       CURRENT STATE               ERROR               PORTS
4alkk6og8pae        nodeapp_web.1       takacsmark/swarm-example:1.0    myvm1               Running
Running about an hour ago
ixjb3ztdumfw        nodeapp_web.2       takacsmark/swarm-example:1.0    myvm2               Running
Running about an hour ago
fyl2c8gw7rqe        nodeapp_web.3       takacsmark/swarm-example:1.0    myvm1               Running
Running about an hour ago
1w8ou7h4g15f        nodeapp_web.4       takacsmark/swarm-example:1.0    myvm2               Running
Running about an hour ago
```

Remove one or more stacks

You can remove your stack with the `docker stack rm` command. Let's not use it right now, we'll still need our running stack.

Manage services

Manage service in your Swarm with the `docker service` command and its subcommands.

Create a service

We used this command before when we created the visualizer service. The command comes in the form of `docker service create` and has many options to fine-tune your service. You can explore the options with `docker service create --help` or use the official reference.

Many options are similar to the options of `docker container run` and you'll find options that are specific to Swarm mode.

List services

We used the `docker service ls` command before to list the services in the Swarm.

List tasks

We used the `docker service ps <service_name>` command before to list the tasks in your Node application service.

Scale a service

We used the `docker service scale` command before to scale our service.

See the logs of a service

Use the `docker service logs` command to see the logs of a service, the options are similar to the options of `docker container logs`.

```
$ docker service logs --tail 1 nodeapp_web
nodeapp_web.1.4alkk6og8pae@myvm1    | Server is running on port 3000
nodeapp_web.3.fyl2c8gw7rqe@myvm1    | Server is running on port 3000
nodeapp_web.2.ixjb3ztdumfw@myvm2    | Server is running on port 3000
nodeapp_web.4.1w8ou7h4g15f@myvm2    | Server is running on port 3000
```

As you can see, the service logs are displayed for all tasks that belong to the service.

Update a service

Use the `docker service update` command to update the configuration of a service. You can change the configuration that you specified with `docker service create` with this command.

Rollback service changes

After you applied changes to your service with the `docker service update` command, you can revert to the previous state with the `docker service rollback` command, if needed.

Manage tasks and containers

As I mentioned before, you do not manage tasks and containers in the Swarm directly. I emphasize this point again, because many experts feel tempted to hack the containers on the lowest level directly when something needs to be fixed asap. If you need to apply changes to your configuration, please reconfigure your services and drive your Swarm with the service configurations.

If you still need to look inside a container to check something during troubleshooting use the regular `docker container` commands.

Manage nodes

Docker Swarm has dedicated commands for node management.

List nodes

Use the `docker node ls` command to list nodes in the Swarm.

```
$ docker node ls
ID                          HOSTNAME            STATUS              AVAILABILITY        MANAGER
STATUS      ENGINE VERSION
yttgwlrspn5ln3ho1qffiy4op *   myvm1               Ready               Active              Leader
18.09.1
tsmjsfj4f9ht5fefxwy2h7nve     myvm2               Ready               Active
18.09.1
```

You can see that both nodes are listed in the command output. It is clear that `myvm1` is the leader and the Docker engine versions are also displayed. Please note the little `*` next to the ID of `myvm1`, the asterisk denotes that this is the active node, in other words my shell commands are connected to this node. Remember, we used the `eval $(docker-machine env myvm1)` command to configure the shell properly.

Add deployment configuration to the Compose file

Let's update our example with Swarm deployment configuration for our services. I'll add the visualizer service to the Compose file, too.

```
version: '3'

services:
    web:
        build: .
        image: takacsmark/swarm-example:1.0
        ports:
            - 80:3000
        networks:
            - mynet
        deploy:
            replicas: 6
            update_config:
                parallelism: 2
                delay: 10s
            restart_policy:
                condition: on-failure

    visualizer:
        image: dockersamples/visualizer:stable
        ports:
          - "8080:8080"
        volumes:
          - "/var/run/docker.sock:/var/run/docker.sock"
        deploy:
          placement:
            constraints: [node.role == manager]

networks:
    mynet:
```

Let's update our stack! Please remove the visualizer service first, because we will recreate it with the use of the Compose file. Use the service name that you get with `docker service ls` in the `docker servicer rm` command. My looks like this:

```
$ docker service rm recursing_visvesvaraya
recursing_visvesvaraya
```

Please note that I added deployment preferences to our Node server; we'll run 6 replicas, I configured the rolling updates to update 2 containers at a time with a 10s delay between update groups and I also configured a restart policy to restart containers on-failure.

It's a common option to specify resources in the Compose file with the `resources` option under `deploy`. You can limit and reserve CPU and memory for your containers with the use of `resources`.

In the visualizer service I have specified the placement of the container with the constraint that is should be placed on a manager node. Docker gives you further options like `node.id`, `node.hostname` to define your constraints, but in real life, you'll probably set up your own labels to be as flexible as possible with your placement. Check out the `node.labels` and `engine.labels` constraints. You can read more about this in the official Docker documentation.

Let's re-deploy our stack.

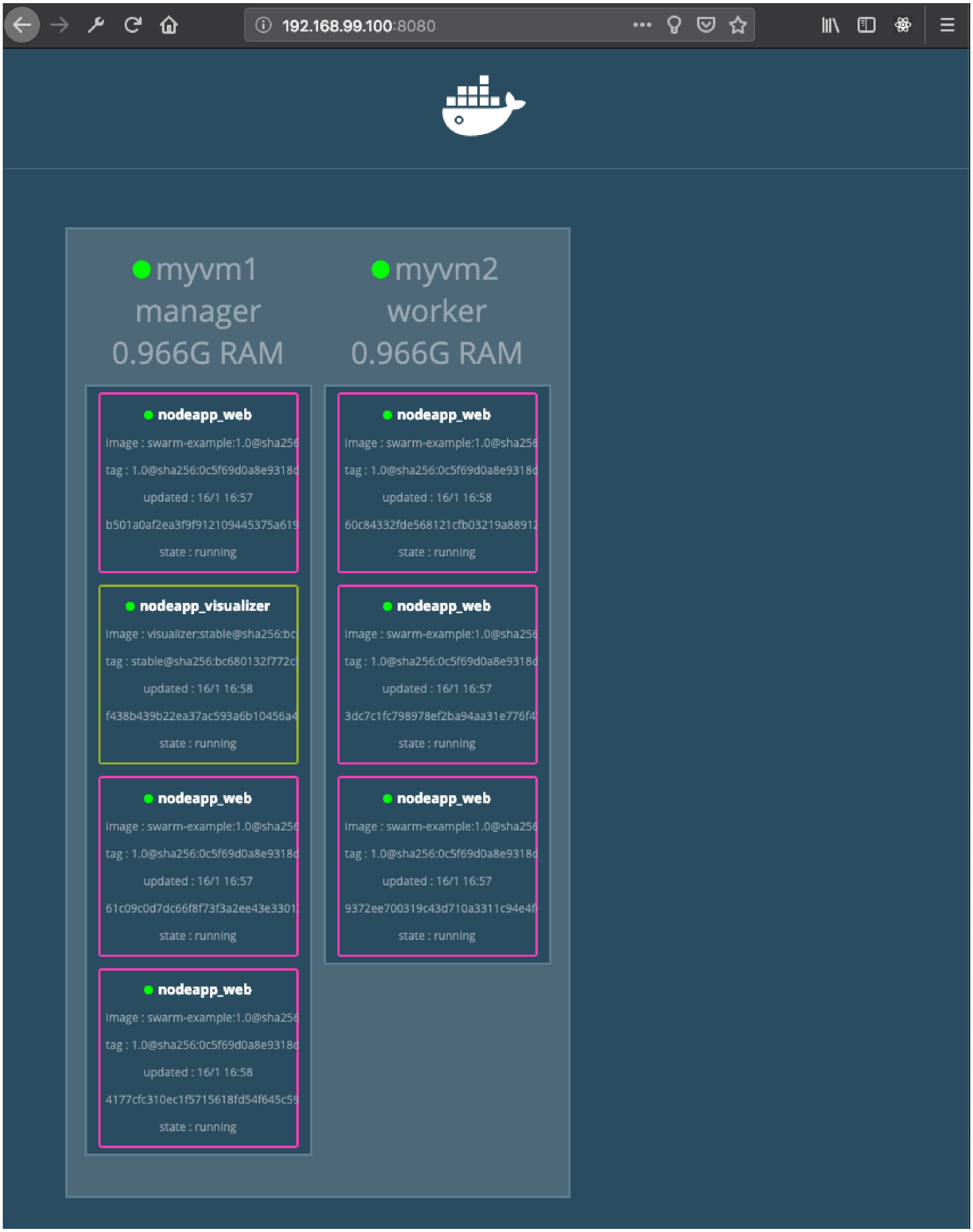
```
$ docker stack deploy nodeapp -c docker-compose.yml
Ignoring unsupported options: build

Creating network nodeapp_default
Creating service nodeapp_visualizer
Updating service nodeapp_web (id: vtj81lstp1lwcexhyhdr02xi1)
```

As you can see, Docker has updated our stack automatically. Let's see what's happening in our stack.

```
$ docker stack ps nodeapp
ID                  NAME                      IMAGE                             NODE
DESIRED STATE       CURRENT STATE                    ERROR                PORTS
iimb6icg7a7j        nodeapp_web.1             takacsmark/swarm-example:1.0      myvm1
Running             Running about a minute ago
ttw0kg14931d        nodeapp_visualizer.1      dockersamples/visualizer:stable   myvm1
Running             Running 45 seconds ago
4alkk6og8pae        nodeapp_web.1             takacsmark/swarm-example:1.0      myvm1
Shutdown            Shutdown about a minute ago
k6eu4bsvju28        nodeapp_web.2             takacsmark/swarm-example:1.0      myvm2
Running             Running about a minute ago
ixjb3ztdumfw         \_ nodeapp_web.2         takacsmark/swarm-example:1.0      myvm2
Shutdown            Shutdown about a minute ago
9pocpcbru0ts        nodeapp_web.3             takacsmark/swarm-example:1.0      myvm1
Running             Running about a minute ago
fyl2c8gw7rqe         \_ nodeapp_web.3         takacsmark/swarm-example:1.0      myvm1
Shutdown            Shutdown about a minute ago
hydrprc6bp3u        nodeapp_web.4             takacsmark/swarm-example:1.0      myvm2
Running             Running about a minute ago
1w8ou7h4g15f         \_ nodeapp_web.4         takacsmark/swarm-example:1.0      myvm2
Shutdown            Shutdown about a minute ago
2kqzdn3gdyvj        nodeapp_web.5             takacsmark/swarm-example:1.0      myvm2
Running             Running about a minute ago
vh6scv44k3d1        nodeapp_web.6             takacsmark/swarm-example:1.0      myvm1
Running             Running about a minute ago
```

We have 6 replicas of our Node application running and one replica of visualizer. Please be patient, it takes some time to apply all the changes.

Let's see it in visualizer.



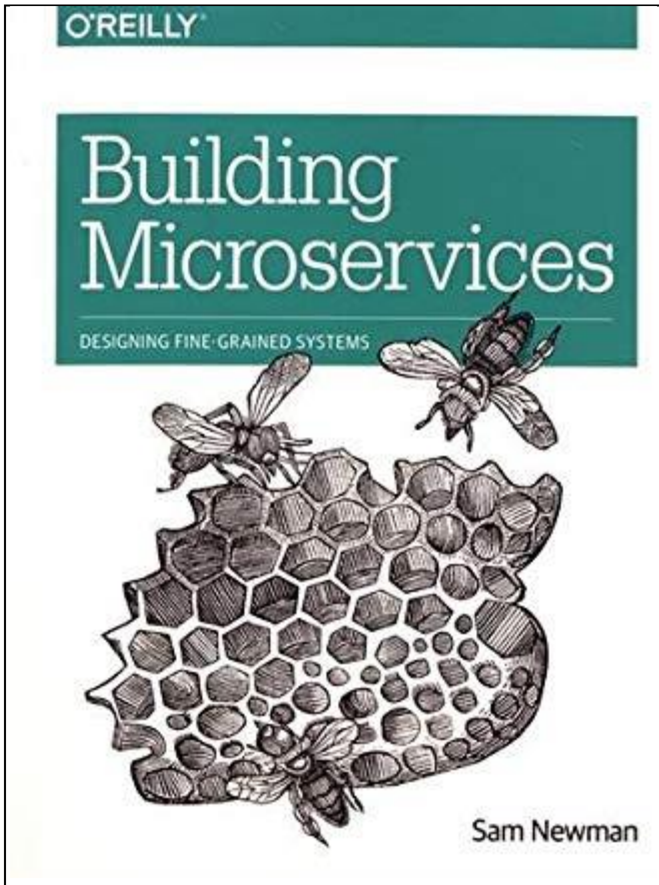
192.168.99.100:8080

myvm1
manager
0.966G RAM

myvm2
worker
0.966G RAM

```
nodeapp_web
image : swarm-example:1.0@sha256
tag : 1.0@sha256:0c5f69d0a8e9318d
updated : 16/1 16:57
b501a0af2ea3f9f912109445375a619
state : running

nodeapp_visualizer
image : visualizer:stable@sha256:bc
tag : stable@sha256:bc680132f772c
updated : 16/1 16:58
f438b439b22ea37ac593a6b10456a4
state : running

nodeapp_web
image : swarm-example:1.0@sha256
tag : 1.0@sha256:0c5f69d0a8e9318d
updated : 16/1 16:57
61c09c0d7dc66f8f73f3a2ee43e3301
state : running

nodeapp_web
image : swarm-example:1.0@sha256
tag : 1.0@sha256:0c5f69d0a8e9318d
updated : 16/1 16:58
4177cfc310ec1f5715618fd54f645c59
state : running
```

```
nodeapp_web
image : swarm-example:1.0@sha256
tag : 1.0@sha256:0c5f69d0a8e9318d
updated : 16/1 16:58
60c84332fde568121cfb03219a88912
state : running

nodeapp_web
image : swarm-example:1.0@sha256
tag : 1.0@sha256:0c5f69d0a8e9318d
updated : 16/1 16:57
3dc7c1fc798978ef2ba94aa31e776f4
state : running

nodeapp_web
image : swarm-example:1.0@sha256
tag : 1.0@sha256:0c5f69d0a8e9318d
updated : 16/1 16:57
9372ee700319c43d710a3311c94e4f
state : running
```



A Great Read - Building Microservices - Designing Fine Grained Systems
sponsored link

Docker has various options for deployment configuration, please refer to the [Compose file reference]() to create the right configuration for your project.

Drain a node

You may want to drain a node in your Swarm to conduct maintenance activities. When you drain a node, Docker will make sure that the node will not receive new tasks and the existing tasks will be rescheduled to active nodes.

Let's check our nodes first.

```
$ docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER
STATUS      ENGINE VERSION
yttgwlrspn5ln3ho1qffiy4op *   myvm1               Ready               Active              Leader
18.09.1
tsmjsfj4f9ht5fefxwy2h7nve     myvm2               Ready               Active
18.09.1
```

Both nodes are active. Let's drain `myvm2`.

```
$ docker node update --availability=drain myvm2
myvm2
```

Let's check the status of the nodes:

```
$ docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER
STATUS      ENGINE VERSION
yttgwlrspn5ln3ho1qffiy4op *   myvm1               Ready               Active              Leader
18.09.1
tsmjsfj4f9ht5fefxwy2h7nve     myvm2               Ready               Drain
18.09.1
```

If you check the visualizer output now, you'll see that all 6 replicas of the Node application are running on the manager node now.

Let's bring the node back to life.

```
$ docker node update --availability=active myvm2
myvm2
```

If you check visualizer now, then you'll notice that all 6 replicas are still running on the manager node. This is because Docker does not automatically re-distribute containers when a node becomes active after maintenance. The newly activated node will receive new tasks, but the old tasks are not rescheduled automatically.

Let's redistribute our containers with the following command.

```
$ docker service update --force nodeapp_web
nodeapp_web
overall progress: 6 out of 6 tasks
1/6: running   [==================================================>]
2/6: running   [==================================================>]
3/6: running   [==================================================>]
4/6: running   [==================================================>]
5/6: running   [==================================================>]
6/6: running   [==================================================>]
verify: Service converged
```

This command gets the job done, but if you have many services, you'll want to write your own script to automate this task at scale.

Rolling update

Let's create a new version of our Node application image. Let's update the image version in our `docker-compose.yml`.

```
        image: takacsmark/swarm-example:1.1
```

Let's build and push the new image to the Docker Hub.

```
$ docker-compose build
$ docker-compose push
```

Start the rolling update by deploying the stack.

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

Docker will start updating the running containers in groups of 2 with 10 second delay as we specified in the Compose file.

You can follow the progress regularly running the command to list the tasks in the Node service. You'll see that the containers are updated in waves as expected.

```
$ docker service ps nodeapp_web
```

Alternatively you can use the `docker service update --image` command to start a rolling update, but I prefer the Compose file variant, because it's more explicit and you can add automation on top of it.

Swarm networking and service discovery

We defined a custom network in our Compose file, and Docker automatically created this network when we deployed our stack. We were able to access our application using the IP address of any of the hosts in the Swarm. Let's see how this works.

Docker has a network driver called the overlay network driver. Overlay networks are networks over multiple Docker host machines. You can connect multiple machines that run the Docker daemon using one overlay network.

When we deployed our stack to the Swarm, our user defined network was created in the form of an overlay network. Overlay is the default driver when you are using the Compose file in Swarm mode.

Swarm services use a solution called the routing mesh for service discovery. This means that if you publish ports in your services, the published ports will be available on all Swarm nodes, and you can connect to a service's published port on any node even if that node is not running a container of the specific service. Your request is redirected to the node that is running the service automatically in the background.

The Swarm implements internal load balancing to distribute requests among replicated services in the cluster.

High availability cluster in the cloud

The two machine Swarm we just set up is for demonstration purposes. Setting up a high availability cluster for production requires careful design and sizing.

If you want to get started with a high availability Swarm setup, you'll find pre-defined templates that you can use with various cloud providers. Let's have a look at the AWS template.

Docker provides a link to the latest stable release CloudFormation template on their website.

You can move on from that page using the “Deploy Docker Community Edition (CE) for AWS (stable)” link. At the time of writing this link points to this URL.

You can use the wizard here to set up a high availability Swarm cluster on AWS.

Please note that the default configuration contains 3 Swam managers, 5 workers, so running this setup continuously will definitely generate charges even if you are using the free tier.

This wizard is a great place to start, if you’d like to follow along a complete step-by-step setup, you can find one in my Docker book.

Other container orchestrator solutions you should be aware of

Now that we have seen the key basics of Docker Swarm, let’s have a look at the broader picture of container orchestration solutions.

Docker Swarm is not your only option for container orchestration, it is not even to most popular solution out there. Let’s see what are the main tasks of an orchestrator.

- Abstract away the underlying infrastructure details.
- Maintain the desired state of your containers in your cluster described in a descriptor file like the Compose file.
- Scale your services in-line with your requirements.
- Secure the communications in the cluster network.
- Allow service discovery in the cluster.
- Rolling update of cluster services.

Besides Swarm, there are multiple orchestrator alternatives on the market that you can choose from. The top 4 products considered by most projects are Google’s Kubernetes, Docker Swarm, Mesos and HashiCorp’s Nomad.

Kubernetes by Google is the market leading container orchestrator solution today. Its wide adoption is understandable thanks to Google’s wast experience in cloud computing and the large community around the product. Kubernetes, or k8s in short, is free and open-source. Kubernetes was open-sourced in 2014 and it is maintained by the Cloud Native Computing Foundation.

Swarm is Docker’s built in orchestrator. Introduced in 2014, Swarm was added to the Docker engine. Due to Kubernetes’ wide adoption, Docker’s support for Kubernetes grows with every release.

Mesos and Nomad also have a strong base, although right now Kubernetes seems to be the winner that takes it all.