

Get Started, Part 4: Swarms

Estimated reading time: 20 minutes

1: Orientation (<https://docs.docker.com/get-started/part1>)

2: Containers (<https://docs.docker.com/get-started/part2>)

3: Services (<https://docs.docker.com/get-started/part3>)

4: Swarms (<https://docs.docker.com/get-started/part4>)

5: Stacks (<https://docs.docker.com/get-started/part5>)

6: Deploy your app (<https://docs.docker.com/get-started/part6>)

Prerequisites

- Install Docker version 1.13 or higher (<https://docs.docker.com/engine/installation/>).
- Get Docker Compose (<https://docs.docker.com/compose/overview/>) as described in Part 3 prerequisites (<https://docs.docker.com/get-started/part3/#prerequisites>).
- Get Docker Machine (<https://docs.docker.com/machine/overview/>), which is pre-installed with Docker for Mac (<https://docs.docker.com/docker-for-mac/>) and Docker for Windows (<https://docs.docker.com/docker-for-windows/>), but on Linux systems you need to install it directly

(<https://docs.docker.com/machine/install-machine/#installing-machine-directly>). On pre Windows 10 systems *without Hyper-V*, as well as Windows 10 Home, use Docker Toolbox (<https://docs.docker.com/toolbox/overview/>).

- Read the orientation in Part 1 (<https://docs.docker.com/get-started/>).
- Learn how to create containers in Part 2 (<https://docs.docker.com/get-started/part2/>).
- Make sure you have published the `friendlyhello` image you created by pushing it to a registry (<https://docs.docker.com/get-started/part2/#share-your-image>). We use that shared image here.
- Be sure your image works as a deployed container. Run this command, slotting in your info for `username` , `repo` , and `tag` :

```
docker run -p 80:80 username/repo:tag
```

, then visit <http://localhost/> .
- Have a copy of your `docker-compose.yml` from Part 3 (<https://docs.docker.com/get-started/part3/>) handy.

Introduction

In part 3 (<https://docs.docker.com/get-started/part3/>), you took an app you wrote in part 2 (<https://docs.docker.com/get-started/part2/>), and defined how it should run in production by turning it into a service, scaling it up 5x in the process.

Here in part 4, you 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.

Set up your swarm

A swarm is made up of multiple nodes, which can be either physical or virtual machines. The basic concept is simple enough: run `docker swarm init` to enable swarm mode and make your current machine a swarm manager, then run `docker swarm join` on other machines to have them join the swarm as workers. Choose a tab below to see how this plays out in various contexts. We use VMs to quickly create a two-machine cluster and turn it into a swarm.

Create a cluster

Local VMs (Mac, Linux, Windows 7 and 8) (/get-started/part4/#local)

Local VMs (Windows 10/Hyper-V) (/get-started/part4/#localwin)

VMS ON YOUR LOCAL MACHINE (MAC, LINUX, WINDOWS 7 AND 8)

You need a hypervisor that can create virtual machines (VMs), so install Oracle VirtualBox (<https://www.virtualbox.org/wiki/Downloads>) for your machine's OS.

Note: If you are on a Windows system that has Hyper-V installed, such as Windows 10, there is no need to install VirtualBox and you should use Hyper-V instead. View the instructions for Hyper-V systems by clicking the Hyper-V tab above. If you are using Docker Toolbox (<https://docs.docker.com/toolbox/overview/>), you should already have VirtualBox installed as part of it, so you are good to go.

Now, create a couple of VMs using `docker-machine` , using the VirtualBox driver:

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

LIST THE VMS AND GET THEIR IP ADDRESSES

You now have two VMs created, named `myvm1` and `myvm2` .

Use this command to list the machines and get their IP addresses.

```
docker-machine ls
```

Here is example output from this command.

```
$ docker-machine ls
NAME      ACTIVE   DRIVER        STATE     URL
myvm1     -        virtualbox    Running   tcp://192.168.99.100:2376
myvm2     -        virtualbox    Running   tcp://192.168.99.101:2376
```

INITIALIZE THE SWARM AND ADD NODES

The first machine acts as the manager, which executes management commands and authenticates workers to join the swarm, and the second is a worker.

You can send commands to your VMs using `docker-machine ssh`. Instruct `myvm1` to become a swarm manager with `docker swarm init` and look for output like this:

```
$ docker-machine ssh myvm1 "docker swarm init --advertise-addr <myvm1 ip>
Swarm initialized: current node <node ID> is now a manager.
```

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

```
docker swarm join \
  --token <token> \
  <myvm ip>:<port>
```

To add a manager to this swarm, run `'docker swarm join-token manager`

✔ Ports 2377 and 2376

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>).

✔ Having trouble using SSH? Try the `--native-ssh` flag

Docker Machine has the option to let you use your own system's SSH (<https://docs.docker.com/machine/reference/ssh/#different-types-of-ssh>), if for some reason you're having trouble sending commands to your Swarm manager. Just specify the `--native-ssh` flag when invoking the `ssh` command:

```
docker-machine --native-ssh ssh myvm1 ...
```

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"
```

This node joined a swarm as a worker.

Congratulations, you have created your first swarm!

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

```
$ docker-machine ssh myvm1 "docker node ls"
ID                                HOSTNAME          STATUS
brtu9urxwfd5j0zrmkubhpkbd        myvm2            Ready
rihwohkh3ph38fhillhbb84sk *      myvm1            Ready
```

✔ Leaving a swarm

If you want to start over, you can run `docker swarm leave` from each node.

Deploy your app on the swarm cluster

The hard part is over. Now you just repeat the process you used in part 3 (<https://docs.docker.com/get-started/part3/>) to deploy on your new swarm. Just remember that only swarm managers like `myvm1` execute Docker commands; workers are just for capacity.

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

So far, you've been wrapping Docker commands in `docker-machine ssh` to talk to the VMs. Another option is to run `docker-machine env <machine>` to get and run a command that configures your current shell to talk to the Docker daemon on the VM. This method works better for the next step because it allows you to use your local `docker-compose.yml` file to deploy the app "remotely" without having to copy it anywhere.

Type `docker-machine env myvm1`, then copy-paste and run the command provided as the last line of the output to configure your shell to talk to `myvm1`, the swarm manager.

The commands to configure your shell differ depending on whether you are Mac, Linux, or Windows, so examples of each are shown on the tabs below.

Mac, Linux (/get-started/part4/#mac-linux-machine)

Windows (/get-started/part4/#win-machine)

DOCKER MACHINE SHELL ENVIRONMENT ON MAC OR LINUX

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

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

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


```
eval $(docker-machine env myvm1)
```

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

```
$ docker-machine ls
NAME      ACTIVE  DRIVER        STATE     URL                                     ;
myvm1     *       virtualbox    Running   tcp://192.168.99.100:2376
myvm2     -       virtualbox    Running   tcp://192.168.99.101:2376
```

Deploy the app on the swarm manager

Now that you have `myvm1`, you can use its powers as a swarm manager to deploy your app by using the same `docker stack deploy` command you used in part 3 to `myvm1`, and your local copy of `docker-compose.yml`. This command may take a few seconds to complete and the deployment takes some time to be available. Use the `docker service ps <service_name>` command on a swarm manager to verify that all services have been redeployed.

You are connected to `myvm1` by means of the `docker-machine` shell configuration, and you still have access to the files on your local host. Make sure you are in the same directory as before, which includes the `docker-compose.yml` file you created in part 3 (<https://docs.docker.com/get-started/part3/#docker-compose.yml>).

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

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

And that's it, the app is deployed on a swarm cluster!

🟢 **Note:** If your image is stored on a private registry instead of Docker Hub, you need to be logged in using `docker login <your-registry>` and then you need to add the `--with-registry-auth` flag to the above command. For example:

```
docker login registry.example.com
```

```
docker stack deploy --with-registry-auth -c docker-compose.yml
```

This passes the login token from your local client to the swarm nodes where the service is deployed, using the encrypted WAL logs. With this information, the nodes are able to log into the registry and pull the image.

Now you can use the same docker commands you used in part 3 (<https://docs.docker.com/get-started/part3/#run-your-new-load-balanced-app>). Only this time notice that the services (and associated containers) have been distributed between both `myvm1` and `myvm2`.

```
$ docker stack ps getstartedlab
```

ID	NAME	IMAGE	NODE	DI
jq2g3qp8nzw	getstartedlab_web.1	john/get-started:part2	myvm1	Ri
88wgshobzoxl	getstartedlab_web.2	john/get-started:part2	myvm2	Ri
vbb1qbkb0o2z	getstartedlab_web.3	john/get-started:part2	myvm2	Ri
ghii74p9budx	getstartedlab_web.4	john/get-started:part2	myvm1	Ri
0prmarhavs87	getstartedlab_web.5	john/get-started:part2	myvm2	Ri

✔ Connecting to VMs with `docker-machine env` and `docker-machine ssh`

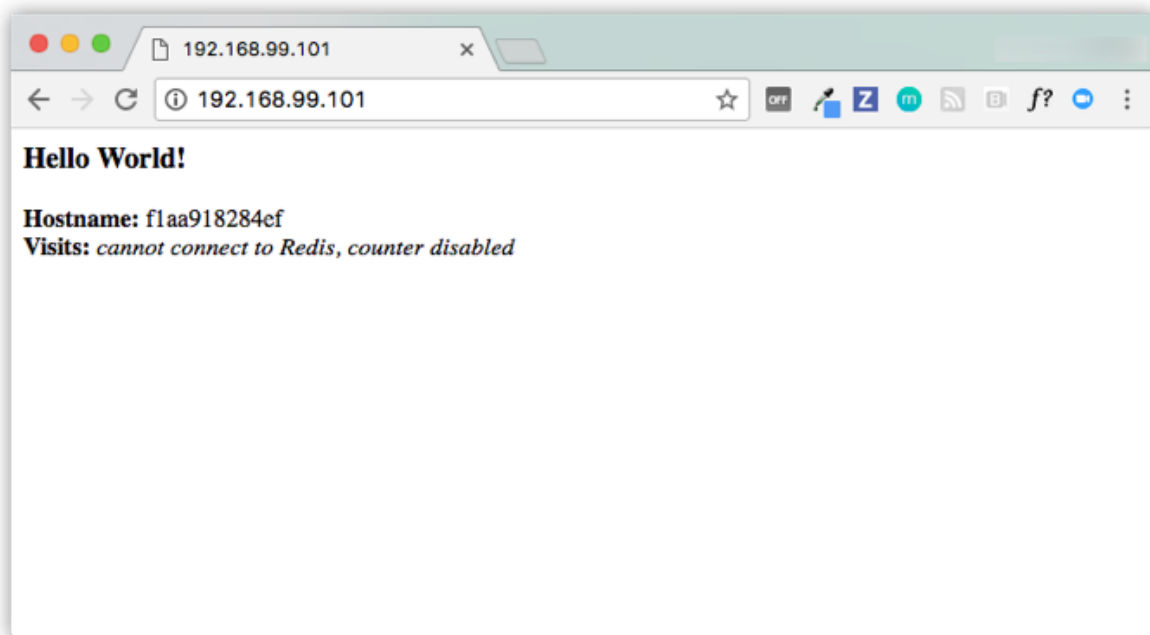
- To set your shell to talk to a different machine like `myvm2`, simply re-run `docker-machine env` in the same or a different shell, then run the given command to point to `myvm2`. This is always specific to the current shell. If you change to an unconfigured shell or open a new one, you need to re-run the commands. Use `docker-machine ls` to list machines, see what state they are in, get IP addresses, and find out which one, if any, you are connected to. To learn more, see the Docker Machine getting started topics (<https://docs.docker.com/machine/get-started/#create-a-machine>).
- Alternatively, you can wrap Docker commands in the form of `docker-machine ssh <machine> "<command>"`, which logs directly into the VM but doesn't give you immediate access to files on your local host.
- On Mac and Linux, you can use `docker-machine scp <file> <machine>:~` to copy files across machines, but Windows users need a Linux terminal emulator like Git Bash (<https://git-for-windows.github.io/>) for this to work.

This tutorial demos both `docker-machine ssh` and `docker-machine env`, since these are available on all platforms via the `docker-machine` CLI.

Accessing your cluster

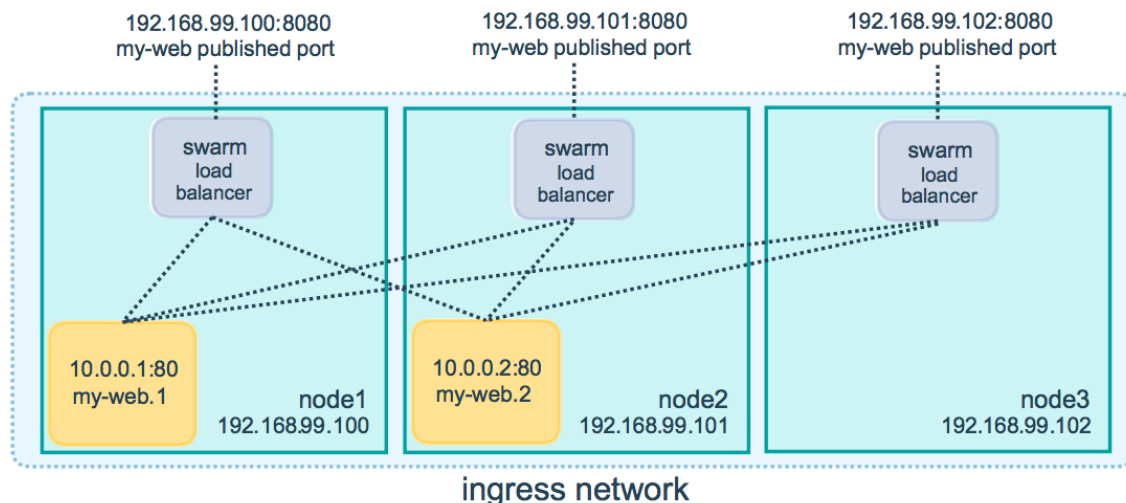
You can access your app from the IP address of **either** `myvm1` or `myvm2`.

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).



There are five possible container IDs all cycling by randomly, demonstrating the load-balancing.

The reason both IP addresses work is that nodes in a swarm participate in an ingress **routing mesh**. This ensures that a service deployed at a certain port within your swarm always has that port reserved to itself, no matter what node is actually running the container. Here's a diagram of how a routing mesh for a service called `my-web` published at port `8080` on a three-node swarm would look:



🔍 Having connectivity trouble?

Keep in mind that to use the ingress network in the swarm, you need to have the following ports open between the swarm nodes before you enable swarm mode:

- Port 7946 TCP/UDP for container network discovery.
- Port 4789 UDP for the container ingress network.

Iterating and scaling your app

From here you can do everything you learned about in parts 2 and 3.

Scale the app by changing the `docker-compose.yml` file.

Change the app behavior by editing code, then rebuild, and push the new image. (To do this, follow the same steps you took earlier to build the app (<https://docs.docker.com/get-started/part2/#build-the-app>) and publish the image (<https://docs.docker.com/get-started/part2/#publish-the-image>)).

In either case, simply run `docker stack deploy` again to deploy these changes.

You can join any machine, physical or virtual, to this swarm, using the same `docker swarm join` command you used on `myvm2`, and capacity is added to your cluster. Just run `docker stack deploy` afterwards, and your app can take advantage of the new resources.

Cleanup and reboot

Stacks and swarms

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

```
docker stack rm getstartedlab
```

✔ Keep the swarm or remove it?

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, but *you need this swarm for part 5, so keep it around for now.*

Unsetting docker-machine shell variable settings

You can unset the `docker-machine` environment variables in your current shell with the following command:

```
eval $(docker-machine env -u)
```

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). To learn more, see the Machine topic on unsetting environment variables (<https://docs.docker.com/machine/get-started/#unset-environment-variables-in-the-current-shell>).

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	ERROR
myvm1	-	virtualbox	Stopped			Unknown	
myvm2	-	virtualbox	Stopped			Unknown	



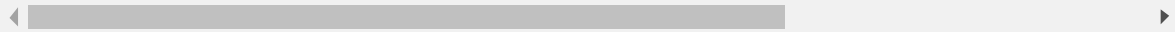
To restart a machine that's stopped, run:

```
docker-machine start <machine-name>
```

For example:

```
$ docker-machine start myvm1
Starting "myvm1"...
(myvm1) Check network to re-create if needed...
(myvm1) Waiting for an IP...
Machine "myvm1" was started.
Waiting for SSH to be available...
Detecting the provisioner...
Started machines may have new IP addresses. You may need to re-run tl

$ docker-machine start myvm2
Starting "myvm2"...
(myvm2) Check network to re-create if needed...
(myvm2) Waiting for an IP...
Machine "myvm2" was started.
Waiting for SSH to be available...
Detecting the provisioner...
Started machines may have new IP addresses. You may need to re-run tl
```



On to Part 5 >> [<https://docs.docker.com/get-started/part5/>]

Recap and cheat sheet (optional)

Here's a terminal recording of what was covered on this page

(<https://asciinema.org/a/113837>):

```
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on thi
bash-3.2$ docker-machine ssh myvm1 "docker swarm init"
Error response from daemon: could not choose an IP address to advertise sinc
1) - specify one with --advertise-addr
exit status 1
bash-3.2$ docker-machine ls
NAME      ACTIVE  DRIVER        STATE     URL                                SWARM
myvm1     -       virtualbox    Running   tcp://192.168.99.104:2376
myvm2     -       virtualbox    Running   tcp://192.168.99.105:2376
bash-3.2$ docker-machine ssh myvm1 "docker swarm init --advertise-addr 192.168.99.104"
Swarm initialized: current node (x500bs7lrweto0jkg6xq2ybd) is now a manager

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

    docker swarm join \
      --token SWMTKN-1-197eaghawf5wqunblowkmgwjov38ugtmscs943xrrz0jk6bpc-4uor
      192.168.99.104:2377
```

To add a manager to this swarm, run 'docker swarm join-token manager' and fo

▶ 00:00

In part 4 you learned what a swarm is, how nodes in swarms can be managers or workers, created a swarm, and deployed an application on it. You saw that the core Docker commands didn't change from part 3, they just had to be targeted to run on a swarm master. You also saw the power of Docker's networking in action, which kept load-balancing requests across containers, even though they were running on different machines. Finally, you learned how to iterate and scale your app on a cluster.

Here are some commands you might like to run to interact with your swarm and your VMs a bit:


```

docker-machine create --driver virtualbox myvm1 # Create a VM (Mac, Linux, Windows)
docker-machine create -d hyperv --hyperv-virtual-switch "myswitch" myvm1 # Create a VM (Hyper-V)
docker-machine env myvm1 # View basic information about the VM
docker-machine ssh myvm1 "docker node ls" # List the nodes in the swarm
docker-machine ssh myvm1 "docker node inspect <node ID>" # Inspect a node
docker-machine ssh myvm1 "docker swarm join-token -q worker" # View the join token for a worker
docker-machine ssh myvm1 # Open an SSH session with the VM; type "exit" to close
docker node ls # View nodes in swarm (while logged on to a VM)
docker-machine ssh myvm2 "docker swarm leave" # Make the worker leave the swarm
docker-machine ssh myvm1 "docker swarm leave -f" # Make master leave the swarm
docker-machine ls # list VMs, asterisk shows which VM this shell is currently logged on to
docker-machine start myvm1 # Start a VM that is currently stopped
docker-machine env myvm1 # show environment variables and commands to run on the VM
eval $(docker-machine env myvm1) # Mac command to connect shell to VM
"C:\Program Files\Docker\Docker\Resources\bin\docker-machine.exe" # Windows command to connect shell to VM
docker stack deploy -c <file> <app> # Deploy an app; command shell is logged on to the VM
docker-machine scp docker-compose.yml myvm1:~ # Copy file to node's local file system
docker-machine ssh myvm1 "docker stack deploy -c <file> <app>" # Deploy an app from the VM
eval $(docker-machine env -u) # Disconnect shell from VMs, use new shell to connect
docker-machine stop $(docker-machine ls -q) # Stop all VMs
docker-machine rm $(docker-machine ls -q) # Delete all VMs and their disks

```

[swarm \(https://docs.docker.com/glossary/?term=swarm\)](https://docs.docker.com/glossary/?term=swarm), [scale \(https://docs.docker.com/glossary/?term=scale\)](https://docs.docker.com/glossary/?term=scale), [cluster \(https://docs.docker.com/glossary/?term=cluster\)](https://docs.docker.com/glossary/?term=cluster), [machine \(https://docs.docker.com/glossary/?term=machine\)](https://docs.docker.com/glossary/?term=machine), [vm \(https://docs.docker.com/glossary/?term=vm\)](https://docs.docker.com/glossary/?term=vm), [manager \(https://docs.docker.com/glossary/?term=manager\)](https://docs.docker.com/glossary/?term=manager), [worker \(https://docs.docker.com/glossary/?term=worker\)](https://docs.docker.com/glossary/?term=worker), [deploy \(https://docs.docker.com/glossary/?term=deploy\)](https://docs.docker.com/glossary/?term=deploy), [ssh \(https://docs.docker.com/glossary/?term=ssh\)](https://docs.docker.com/glossary/?term=ssh), [orchestration \(https://docs.docker.com/glossary/?term=orchestration\)](https://docs.docker.com/glossary/?term=orchestration)