Introduction

In part 3, we scale our application and enable load-balancing. To do this, we must go one level up in the hierarchy of a distributed application: the **service**.

* Stack
* **Services** (you are here)
* Container (covered in [part 2](https://docs.docker.com/get-started/part2/))

About services

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.

Your first docker-compose.yml file

A docker-compose.yml file is a YAML file that defines how Docker containers should behave in production.

docker-compose.yml

Save this file as docker-compose.yml wherever you want. Be sure you have [pushed the image](https://docs.docker.com/get-started/part2/#share-your-image) you created in [Part 2](https://docs.docker.com/get-started/part2/) to a registry, and update this .yml by replacing 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

resources:

limits:

cpus: "0.1"

memory: 50M

restart\_policy:

condition: on-failure

ports:

- "80:80"

networks:

- webnet

networks:

webnet:

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

* Pull [the image we uploaded in step 2](https://docs.docker.com/get-started/part2/) 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 80 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

Before we can use the docker stack deploy command we first run:

docker swarm init

**Note**: We get into the meaning of that command in [part 4](https://docs.docker.com/get-started/part4/). If you don’t run docker swarm init you get an error that “this node is not a swarm manager.”

Now let’s run it. You need to give your app a name. Here, it is set to getstartedlab:

docker stack deploy -c docker-compose.yml getstartedlab

Our single service stack is running 5 container instances of our deployed image on one host. Let’s investigate.

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 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).

**Running Windows 10?**

Windows 10 PowerShell should already have curl available, but if not you can grab a Linux terminal emulator like [Git BASH](https://git-for-windows.github.io/" \t "_blank), or download [wget for Windows](http://gnuwin32.sourceforge.net/packages/wget.htm) which is very similar.

**Slow response times?**

Depending on your environment’s networking configuration, it may take up to 30 seconds for the containers to respond to HTTP requests. This is not indicative of Docker or swarm performance, but rather an unmet Redis dependency that we address later in the tutorial. For now, the visitor counter isn’t working for the same reason; we haven’t yet added a service to persist data.

Scale the app

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 stack deploy -c docker-compose.yml getstartedlab

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.

Take down the app and the swarm

* Take the app down with docker stack rm:
* docker stack rm getstartedlab
* Take down the swarm.

docker swarm leave --force