3. Services and Stacks

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://getliner

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 (https://en.wikipedia.org/wiki/YAML)** file that **defines (https://docs.docker.com/compose/compose-file/)** 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/engine/reference/commandline/push/)** you created in **Part 2** to a registry, and then update your new YAML file 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:
      - "4000:80"
    networks:
      - webnet
networks:
  webnet:
```

(https://getliner

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

- Pull the image we uploaded in **step 2** from the registry.
- Run 5 instances of that image as a service called *web*, limiting each one to use, at most, 10% of a single core of CPU time (this could also be e.g. “1.5” to mean 1 and half core for each), and 50MB of RAM.
- Immediately restart containers if one fails.
- Map port 4000 on the *host* to port 80 on the *web* service.
- 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**: The Docker `swarm` command is used when configuring a collection of Docker hosts in order to run a collection of containers or services in a highly-available configuration. That topic is beyond the scope of this tutorial, so for now, just accept that this command must be run in order to use the `stack` and `service` commands. If you don't run `docker swarm init` first you will 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 `tutorial`:

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

(https://getliner

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 `tutorial_web`. The service ID is listed as well, along with the number of replicas, image name, and exposed ports.

Alternatively, you can run `docker stack services`, followed by the name of your stack. The following example command lets you view all services associated with the tutorial stack:

```
$ docker stack services tutorial

ID            NAME          MODE          REPLICAS      IMAGE          PORTS
bqpve1djnk0x  tutorial_web  replicated    5/5           username/repo:tag  *:4000->80/tcp
```

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 tutorial_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
```

Try this next: 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 chose—in a round-robin fashion—to respond. The container IDs match your output from the previous command (`docker container ls -q`).

(https://getliner

To view all tasks in a stack, you can run `docker stack ps` followed by your app name, as shown in the following example:

```
$ docker stack ps tutorial
```

ID	NAME	IMAGE	NODE	DESIRED STATE	CURRENT STATE	ERROR	PORTS
uwiaw67sc0eh	tutorial_web.1	username/repo:tag	docker-desktop	Running	Running 9 minutes ago		
sk50xbhmcae7	tutorial_web.2	username/repo:tag	docker-desktop	Running	Running 9 minutes ago		
c4uuw5i6h02j	tutorial_web.3	username/repo:tag	docker-desktop	Running	Running 9 minutes ago		
0dyb70ixu25s	tutorial_web.4	username/repo:tag	docker-desktop	Running	Running 9 minutes ago		
aocrb88ap8b0	tutorial_web.5	username/repo:tag	docker-desktop	Running	Running 9 minutes ago		

> ***NOTE**: 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 (say, from 5 to 3) in `docker-compose.yml`, saving the change, and re-running the docker stack deploy command:

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

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 tutorial
```

(https://getliner

Take down the swarm.

```
$ docker swarm leave --force
```

It's as easy as that to stand up and scale your app with Docker. You've taken a huge step towards learning how to run containers in production.

Recap and cheat sheet

To recap, while typing docker run is simple enough, the true implementation of a container in production is running it as a *service*, typically as part of a *stack*. Services codify a container's behavior in a Compose file, and this file can be used to scale, limit, and redeploy our app. Changes to the service can be applied in place, as it runs, using the same command that launched the service: `docker stack deploy`.

Some commands to explore at this stage:

```
docker stack ls                                # List stacks or apps
docker stack deploy -c <composefile> <appname>  # Run the specified Compose file
docker service ls                 # List running services associated with an app
docker service ps <service>                  # List tasks associated with an app
docker inspect <task or container>                  # Inspect task or container
docker container ls -q                                # List container IDs
```

```
docker stack rm <appname>                    # Tear down an application

docker swarm leave --force      # Take down a single node swarm from the manager
```

(https://getliner