

```

{
  "total": 5,
  "size": 277715,
  "last_used": "2018-04-13T20:30:53.000Z",
  "uptime": 259.215
}

```

We can now manage our container in a similar fashion to the service management on the current operative systems. You can start, stop, and restart it:

```

1. base (bash)
> docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
448c5ddd15f4        imaginini:0.0.5   "node /opt/app/imagine..."   40 minutes ago    Up 40 minutes     0.0.0.0:80->3000/tcp   keen_lalande
> docker stop 448c5ddd15f4
448c5ddd15f4
> docker start 448c5ddd15f4
448c5ddd15f4
> docker restart 448c5ddd15f4
448c5ddd15f4
> docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
448c5ddd15f4        imaginini:0.0.5   "node /opt/app/imagine..."   41 minutes ago    Up 3 seconds      0.0.0.0:80->3000/tcp   keen_lalande
>

```

Every time you change one or more container's states, Docker returns the container IDs that have changed state. This allows you to, for example, to script actions and pipe the output of one command to the next.

## Cleaning containers

Containers, when stopped, are not removed by default. Let's stop our container and list the currently running containers:

```

1. base (bash)
> docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
448c5ddd15f4        imaginini:0.0.5   "node /opt/app/imagine..."   About an hour ago   Up 5 minutes     0.0.0.0:80->3000/tcp   keen_lalande
> docker stop 448c5ddd15f4
448c5ddd15f4
> docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
>

```

Nothing running now. So, where are they? Well, the container is stopped and, while it's not using your processor and memory right now, it's using space. And it's not just that container, either; it's all the containers we ran before.

To see all containers, use the `docker ps -a` command:

COLUMN	CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
	448c5dddf5f4	imagini:0.0.5	"node /opt/app/imagi..."	About an hour ago	Exited (0) 3 minutes ago		keen_lalande
	76f51f26869c	imagini:0.0.3	"node /opt/app/imagi..."	2 hours ago	Exited (137) About an hour ago		epic_kowalevski
	7692dc1ced76	imagini:0.0.3	"node /opt/app/imagi..."	2 hours ago	Exited (1) 2 hours ago		unruffled_cori
	2703b7534d7c	imagini:0.0.3	"node /opt/app/imagi..."	2 hours ago	Exited (1) 2 hours ago		vibrant_minsky
	9fc5faccf333	imagini:0.0.3	"node /opt/app/imagi..."	2 hours ago	Exited (137) About an hour ago		flamboyant_brown
	5f4f0f0b09f2	imagini:0.0.3	"node /opt/app/imagi..."	2 hours ago	Exited (1) 2 hours ago		stupefied_mestorf
	9c2c6c605513	imagini:0.0.2	"node /opt/app/imagi..."	3 hours ago	Exited (1) 3 hours ago		elastic_einstein
	5de2673376c3	cdbef5185faf4	"/bin/sh -c 'apk add..."	28 hours ago	Exited (1) 28 hours ago		dazzling_noether
	3534b6ec35ce	edb7c74921c5	"/bin/sh -c 'npm i..."	28 hours ago	Exited (1) 28 hours ago		peaceful_murdock
	dc6775bad1e8	b211572607c9	"/bin/sh -c 'npm i..."	28 hours ago	Exited (1) 29 hours ago		boring_ardinghell
	cc2e8caa7453	imagini:0.0.1	"sh"	29 hours ago	Exited (0) 29 hours ago		compassionate_torvalds
	4257f3576a16	imagini:0.0.1	"node /opt/app/imagi..."	29 hours ago	Exited (1) 29 hours ago		focused_edison
	178983c5a5e3	imagini:0.0.1	"node /opt/app/imagi..."	29 hours ago	Exited (1) 29 hours ago		pedantic_perlmutter

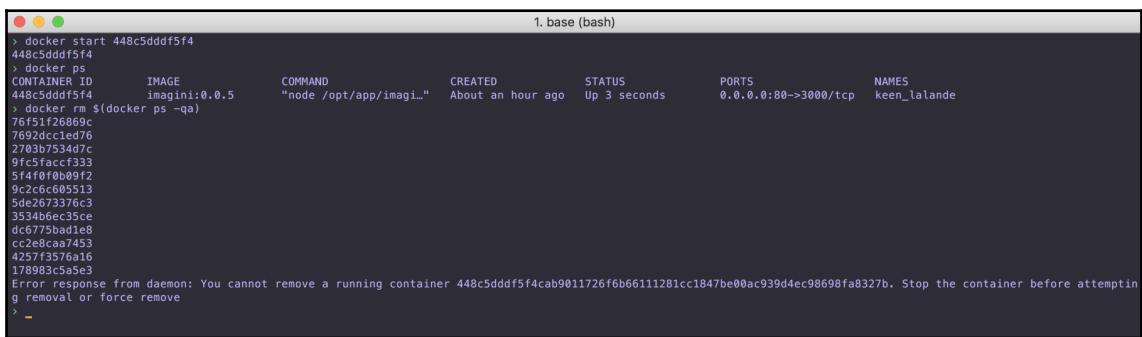
Wow, that's a lot of containers. We can see that we have containers for practically all the images we built, and that means the images are also available and using our disk space:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
imagini	0.0.5	6a0f18cd9ce4	About an hour ago	720MB
imagini	0.0.3	7656998b3a1b	2 hours ago	720MB
imagini	0.0.2	304c4e53de6d	28 hours ago	720MB
<none>	<none>	edb7c74921c5	29 hours ago	107MB
<none>	<none>	b211572607c9	29 hours ago	68.4MB
imagini	0.0.1	a3e21d6fd379	29 hours ago	110MB
node	alpine	7af437a39ec2	2 weeks ago	68.4MB
node	latest	aa3e171e4e95	2 weeks ago	673MB

In the case of images, we can see the size they are occupying. Since we don't need them any more, as they were non-functioning experiences, we can remove them. To do that, we need to use the `rmi` command and pass the image IDs, or the name and version (the TAG column), as parameters.

But before removing the images, we need to remove the containers, or Docker will deny removing them as they're being used. As a dirty workaround, we can use a useful parameter from `docker ps`, which returns only the container IDs and passes those to the remove command. Docker won't remove any running containers, so it's safe to run.

Start our latest container (`docker start`) and then list the containers (`docker ps`):



```
1. base (bash)
> docker start 448c5dddf5f4
448c5dddf5f4
> docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
448c5dddf5f4      imaginini:0.0.5   "node /opt/app/imagine..."   About an hour ago   Up 3 seconds       0.0.0.0:80->3000/tcp   keen_lalande
...
60f5126869c6
7692dc1c1ed76
2703b7534d7c
91c5faccf333
514f0f0b009f2
9c2c6c605513
5de2673376c3
3534b6ec35ce
dc6775bad1e8
cc2e8caa7453
4257f3576a16
178983c5a5e3
Error response from daemon: You cannot remove a running container 448c5dddf5f4cab9011726f6b6611281cc1847be00ac939d4ec98698fa8327b. Stop the container before attempting removal or force remove
> -
```

As described earlier, Docker will indicate each of the container IDs as it changes state (in this case, it removes them). The final error is our running container, which, as I explained earlier, Docker won't allow to be removed as it's running.

We can now do the exact same thing with the images. Those quickly occupy a lot of space and, when developing, it's easy to make changes to images and forget about non-functional images that we left behind:

```
● ● ● 1. base (bash)
> docker rmi $(docker images -q)
Untagged: imaginio:0.0.3
Deleted: sha256:7656998b3a1bfa82f2b69c8476be73c62a5e59e7207fc96f14098d36ced8876
Deleted: sha256:c1ec0df9147e6d194e1107d84088734b8ff690ebba26fe36eba7bfef73e299f7
Deleted: sha256:3b4bd6df57425f866c699d1c8a0aeaf7c3e2c22b538fd5de4415bbb7e4e
Deleted: sha256:9e605c9efc166a380cc47da49da4ee931ad49bb95d270ba45ef2a6e3d05622e98
Deleted: sha256:3781eaaaf2e9876c8d064b4a00eeab7a86eae7f0e3056cc496d53047747536
Deleted: sha256:76fa043ef4ad4eefca9d88b7f264aad9ba1f1441d60518b3f8621e70286ccc08a1b
Untagged: imaginio:0.0.2
Deleted: sha256:304c4e53deed35946d713cac1315b884e29ae3219aaeae0f30109f6e7a35dce
Deleted: sha256:ec99ff91890d57bbff989294526a6cbcddbe94ffdb5b833260b4852f120f6e4a92
Deleted: sha256:7239bf173f308858618ce88d820944e7c970576c90bds13a9dc862b83cf0fe
Deleted: sha256:a6e1d10270a146d561ecd53783c1407d00f276b52e8c240285c2ddd64049c5
Deleted: sha256:c9e587d10b0197113b1600f91e694286705343cb60ecb829d8922db8410c3456
Deleted: sha256:1713f079b632e1a3a7ca855e174a22c274a19b3aa34ce71ebde372f1890a8f
Deleted: sha256:f42c28046424ca7593c4108d59a4bd78ee788a491a57e762213ce512332c2c0
Deleted: sha256:4819a1ef7fb168a7ba13d57b1107d088c75b419e8517310c7f433b2bd45d
Deleted: sha256:996681aef81f3ea3bz2a5d7dab52a42e291fc2a6946fd89ca1bcd861ea6007
Deleted: sha256:c50581b757916502620268e5c369674f8ab298fcc44d5f01f0raaf834789a
Deleted: sha256:edb7c94921c5508a613277f15422c537c8133f5383f7e1d42c37b3b8c422bad08
Deleted: sha256:91fb995cc465b32cb85690a6819b104eed2dade40811d1e15bfafe9a80f135
Deleted: sha256:0b7a4d4c5b95e7b66b85a417d86376ad2275e50007bf8812fe821d975871288b
Deleted: sha256:6af0d8765803d4903d904625b863e166ae043859d0d6f96a185f866ceeb949c1
Deleted: sha256:aidd12597679434e42c9a9164ab1b13dc74913fd6b6212863d20ff039ba78c7
Deleted: sha256:9ba68d9fbba4f811c304353b9063973a482d793130750b8869a99d166
Deleted: sha256:cbd529a453427941fc3c1502b1d0f562bc7f46cd0dee4a4e87dbs5c13d9
Deleted: sha256:0db53cbe1fb2320eb4887a2bed7abce432b541ea5f0ebc9d28c0fd71036c8
Deleted: sha256:5c4e1ebc58b64379efb37e6e94564f5e0125143ada86ab0717cf2d27f7e
Deleted: sha256:b2115726079e2da241de8d9ccb2838ebc981f1518caba14a8263e90bbff1c1c0
Deleted: sha256:9fbe4cd1baca52c782dd93ebf75a4697e552b2fa8ad8e66d8d9d4ca48ae9c993
Deleted: sha256:3c9688fa361leaf51ca93ae1dc2a3d0e21328c174c6ff023b42e48a0e4a29148
Deleted: sha256:10b36c1eb29eaff45410c546543d380adad3e6098623d25ed267f2263a1e84
Deleted: sha256:2ed142e00533cd1cd93749f31c39c4537103102966c84616206b57e3d6a678
Deleted: sha256:9d933e38b8cedf88c9934575254961dc2153b5b5a6fb8f0e8f79c25d2ac5f
Deleted: sha256:cab1931964429883a748c0a64a31bce472a7db8a7382ccefcb02243f744ca8
Untagged: imaginio:0.0.1
Deleted: sha256:a3e21d6fd3796cf5dc475cf6a7c938c023ebd893f72f504a85db6370a9caf
Deleted: sha256:b2115726079e2da241de8d9ccb2838ebc981f1518caba14a8263e90bbff1c1c0
Deleted: sha256:9a41633e305ad6f2702a00458e5b30abe8ce5ee0bd42b2bd095591262ad789
Deleted: sha256:e39pdfbf1fa5e5b9d8f1b031bf081046f11fe2e5818e728e96e27ae3f5698b
Deleted: sha256:9c062473535a2c2675492394e5e9f810d9e313dc926e140e9b299df709900ff
Deleted: sha256:6c7374ffe0c15216e0a2dc93bb4954703e51c8881239bd0ce7df5f97993c9
Deleted: sha256:cdbe5185taf433675e2405b7467617f7f6dae5abbd9fd17583bde2
Untagged: node:alpine
Untagged: node@sha256:5149a828f508d48998e6230cdc8e6832cba192088b442c8effe23d3f3c6892cd3
Deleted: sha256:7a437a39ec2ef08ff95c9dbb766402b31e702544835de18d9263cea18b7c539
Deleted: sha256:69fb42323cd9ff16e32d5b665d78f039b997546c6a8ff9ad00788ff5e62adc
Deleted: sha256:5989e684a4fb3de17cd57e518694a93ef1e78b7b1a1bd1bda5b94eb64d8a
Deleted: sha256:9dfa40a0da3bla8a7c34abc596d81de2db4ecdc0a7211086d66685da1ce6ef
Error response from daemon: conflict: unable to delete 6a0f18cd9c4 (cannot be forced) - image is being used by running container 448c5dddf5f4
Error response from daemon: conflict: unable to delete aa3e171e4e95 (cannot be forced) - image has dependent child images
>
```

The last two errors are expected. First, one of the images is from our running container. Second, the other image is the base image of our image.

You may also notice that there are way more lines than images from the previous list. They're not images; they're the layers of each image (remember the cached steps when building?). Docker stores images in layers, which means that a similar image may have layers in common, and, in turn, some disk space may be saved.

You should now have something like this:

```
1. base (bash)
> docker images
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
imagini         0.0.5   6a0f18cd9ce4  About an hour ago  720MB
node            latest   aa3e171e4e95  2 weeks ago   673MB
> -
```

That looks much better; no more garbage from our previous development iterations. We can now move on and focus on another dependency of our service that we still rely on and is not on a container yet.

## Deploying MySQL

We now have our service running inside a container, with all its dependencies properly defined and installed inside. But we still need a database server and we're still relying on MySQL, which is on our host.

A single container was not meant to run many services at once. Similar to microservices, a container should do only one job. But containers can communicate with the outside, so this means they can communicate with each other.

We can deploy an additional container to run our database server. There are official MySQL container images, so it's as easy as a simple command to start running a database. But first, we need to take care of two things:

- The database server needs to store the database content on the host, or we'll lose the data when we remove our deployment
- Our initial container needs to know where the database server is, and this changes dynamically on each deploy, so we need to have a way of knowing this

Docker has a way of knowing what host port was associated to a container port. This is by using Docker port, and its syntax is simple; you indicate the container to get all port assignments or you indicate a specific port to get only that one:

```
1. base (bash)
> docker ps
CONTAINER ID        IMAGE           COMMAND       CREATED          STATUS          PORTS          NAMES
448c5dddf5f4        imagini:0.0.5   "node /opt/app/imagini..."   2 days ago       Up 16 seconds   0.0.0.0:80->3000/tcp   keen_lalande
> docker port 448c5dddf5f4 3000/tcp
0.0.0.0:80
```

In the case of our original container, we know that we are associated with port 80, and that's what the command indicates. In a production environment, you'll have several containers competing for the same ports. Docker has networks to help us isolate container groups easily and allow us to create custom networks of containers.

Let's just dive in and create a network for our service using `docker network`:

```
1. base (bash)
> docker network create imagin
e30e9bf099c9b092e76e3745d5b8f56810700f85ff2e26f8d8872ad5435ee948
> -
```

That's it. We have a network named `imagin` with the ID returned by the command. We can get a list of current Docker networks with the given command:

```
1. base (bash)
> docker network create imagin
e30e9bf099c9b092e76e3745d5b8f56810700f85ff2e26f8d8872ad5435ee948
> docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
1771243557ee   bridge    bridge      local
3dae39facc82   host      host       local
e30e9bf099c9   imagin    bridge      local
8d55f41d41fb   none     null       local
> -
```

There are now four networks, one of which is the one we created. The other three are:

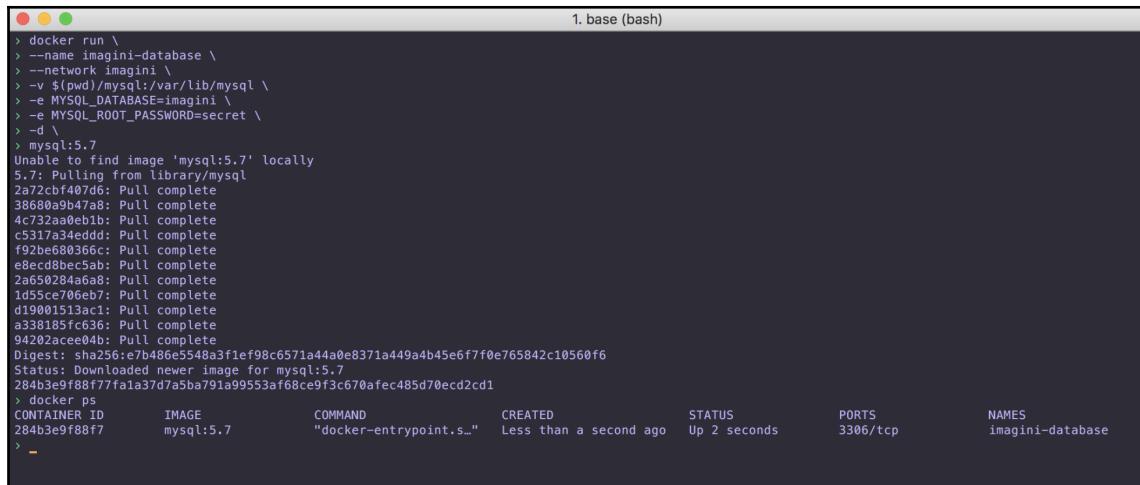
- **Bridge**: The default one for new containers
- **Host**: If you want to attach a container directly to the host network
- **None**: If you don't want a container to have a network at all

Now, let's remove our current deployment and make a few changes. First, let's remove and stop the container:

```
1. base (bash)
> docker ps
CONTAINER ID      IMAGE      COMMAND      CREATED      STATUS      PORTS      NAMES
a7ed0df2fe2b      imagin:0.5   "node /opt/app/imagin..."  3 seconds ago   Up 2 seconds   0.0.0.0:80->3000/tcp   reverent_goldwasser
> docker stop a7ed0df2fe2b && docker rm a7ed0df2fe2b
a7ed0df2fe2b
a7ed0df2fe2b
> -
```

Second, let's deploy our new database server container on our new network. We'll use the official container image. We'll also create a `mysql` folder to store the database and anything else the server needs to operate between deployments without losing information.

I'll also introduce another option from Docker, which enables us to name a container. You might have noticed some funny names when you previously listed containers. We can avoid those random names and use our own:



The screenshot shows a terminal window titled "1. base (bash)". The user runs the command `docker run` with various flags to pull the MySQL image and create a container named `imaginidata`. The output shows the MySQL image being pulled from the library, and the container is created with the name `imaginidata`. Finally, the `ps` command is run to list the running containers, showing the MySQL container with its details.

```
> docker run \
>   --name imaginidata \
>   --network imaginini \
>   -v $(pwd)/mysql:/var/lib/mysql \
>   -e MYSQL_DATABASE=imaginini \
>   -e MYSQL_ROOT_PASSWORD=secret \
>   -d \
>   mysql:5.7
Unable to find image 'mysql:5.7' locally
5.7: Pulling from library/mysql
2a72cbf407d6: Pull complete
38680a9b47a8: Pull complete
4c732aa0eb1b: Pull complete
c5317a34e0dd: Pull complete
f92be680366c: Pull complete
e8ecd8bec5ab: Pull complete
2a650284a6a8: Pull complete
1d55ce706eb7: Pull complete
d19001513aca1: Pull complete
a338185fc636: Pull complete
94202aceee04b: Pull complete
Digest: sha256:e7b486e5548a3f1ef98c6571a44a0e8371a449a4b45e6f7f0e765842c10560f6
Status: Downloaded newer image for mysql:5.7
284b3e9f88f77fa1a37d7a5ba791a99553af68ce9f3c670afec485d70ecd2cd1
> docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
284b3e9f88f7        mysql:5.7          "docker-entrypoint.s..."   Less than a second ago   Up 2 seconds      3306/tcp            imaginidata
> -
```

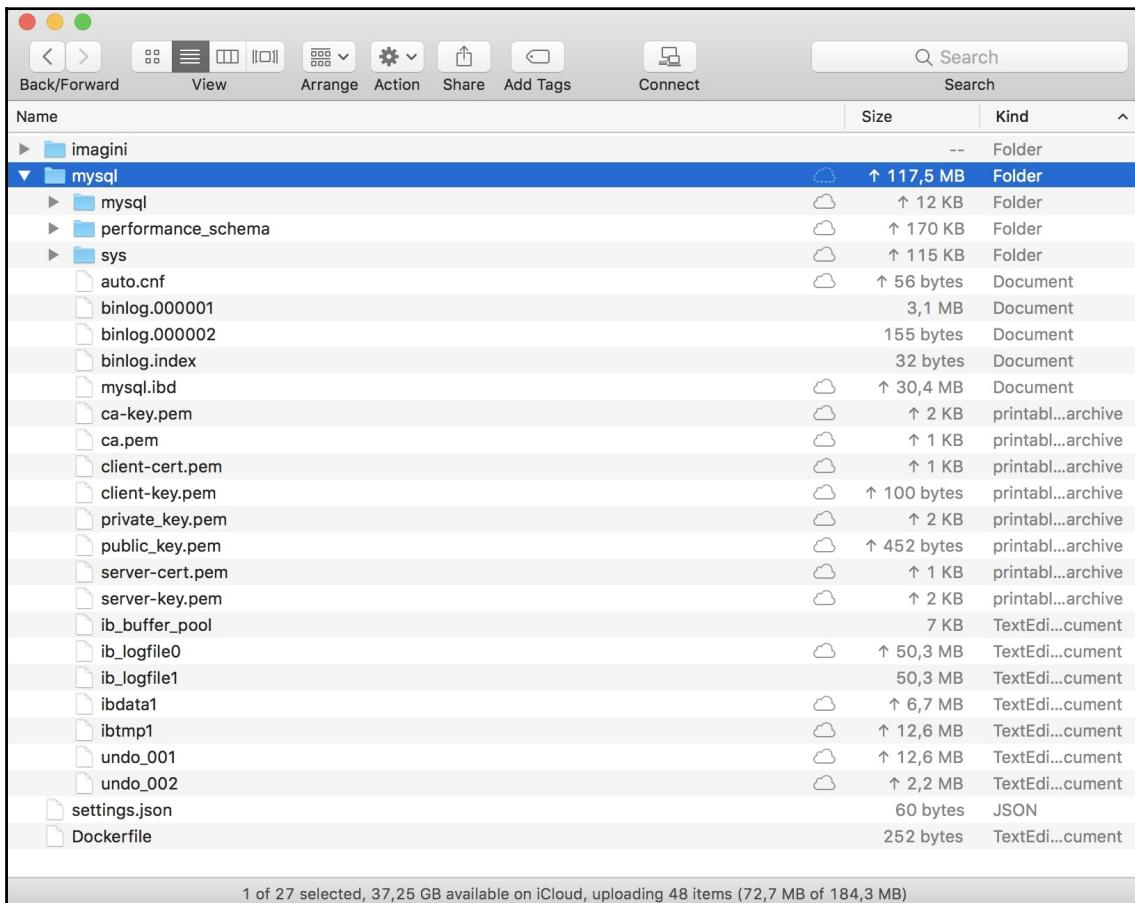
Docker automatically downloaded the latest MySQL version. We now have a database server called `imaginidata` running on our `imaginini` network. Notice we didn't assign any of its ports to the host because we actually don't need to access it outside our custom network.

You can also see that I defined the database to be created initially, as well as the root password. We're specifying image `mysql:5.7`, which points to the latest revision of MySQL version 5.7.



Always remember to use specific versions with containers. Never use the latest version as this may change between development and production, and you need to be certain of the versions you're running.

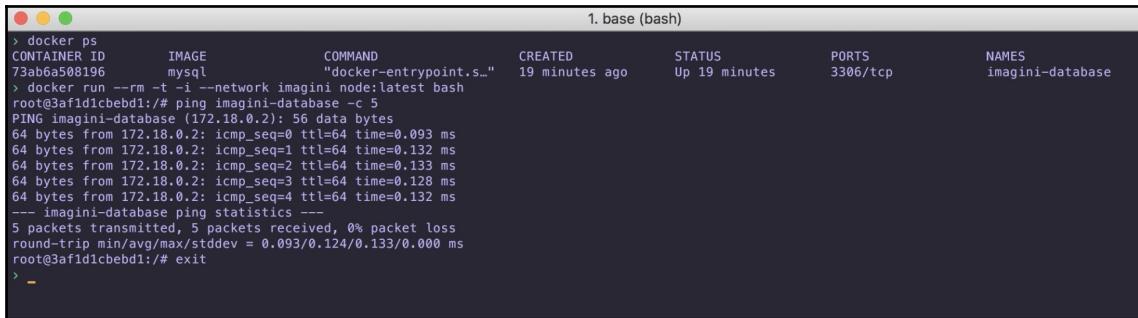
This is specific to this image and you can read more about it on the official Docker Hub page. We can confirm that the server is up and running using our local folder by just looking at its content:



Before starting our main service container, we need to make just a few more changes. First, we need to change the `settings` file to point the configuration to our new database location.

One advantage of using a proper name for a container is that you can use that name just like it was a DNS name. We can confirm that our database server is reachable from another container we created on the same network.

Let's create a container to test this:



```
1. base (bash)
> docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
73ab6a508196        mysql               "docker-entrypoint.s..."   19 minutes ago    Up 19 minutes      3306/tcp            imagini-database
> docker run --rm -t -i --network imagini node:latest bash
root@3afidicbebd1:/# ping imagini-database -c 5
PING imagini-database (172.18.0.2): 56 data bytes
64 bytes from 172.18.0.2: icmp_seq=0 ttl=64 time=0.093 ms
64 bytes from 172.18.0.2: icmp_seq=1 ttl=64 time=0.132 ms
64 bytes from 172.18.0.2: icmp_seq=2 ttl=64 time=0.133 ms
64 bytes from 172.18.0.2: icmp_seq=3 ttl=64 time=0.128 ms
64 bytes from 172.18.0.2: icmp_seq=4 ttl=64 time=0.132 ms
--- imagini-database ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.093/0.124/0.133/0.000 ms
root@3afidicbebd1:/# exit
> -
```

We just used an image that's already available and started the container with a bash console. We tried to ping our database server and it works. We then exited the container. Because we used the `--rm` command, the container is removed after the first stop.

Now, let's remove the settings file to point to the new location and to also change the root password:

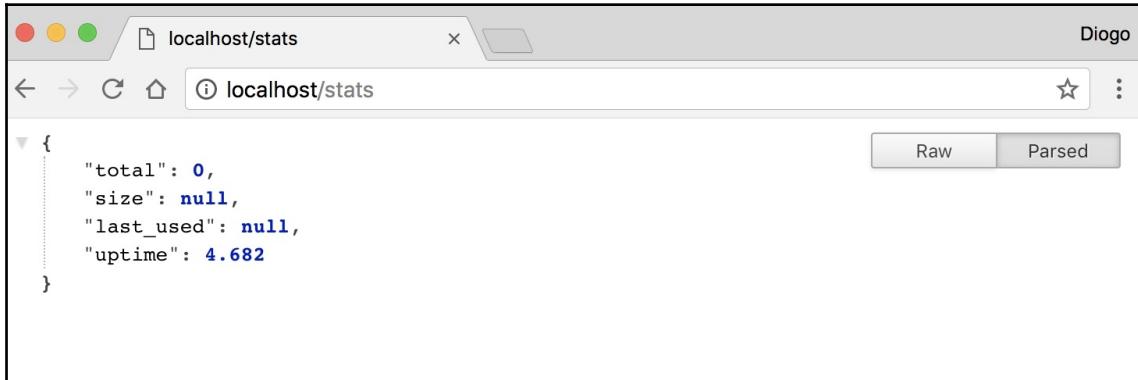
```
{
  "db": "mysql://root:secret@imagini-database/imagini"
}
```

We can now deploy our container again. We'll just give this container a name, too:



```
1. base (bash)
> docker run \
>   --name imagini-service \
>   --network imagini \
>   -p 80:3000 \
>   -d \
>   -v $(pwd)/settings.json:/opt/app/settings.json \
>   imagini:0.5
1a078ec2e4bb6ca6e763a3e5241f55898e9afe2a4aebf3a97ded9fd9dd31644
> docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
1a078ec2e4bb        imagini:0.5      "node /opt/app/imagi..."   Less than a second ago   Up 3 seconds      0.0.0.0:80->3000/tcp   imagini-service
284b3e9f88f7        mysql:5.7       "docker-entrypoint.s..."   About a minute ago    Up About a minute   3306/tcp            imagini-database
> -
```

We can see that our service is running correctly, with connections to the new database server by just hitting the stats address and seeing that there are no images there, contrary to our previous version that we used to test this:



```
{  
  "total": 0,  
  "size": null,  
  "last_used": null,  
  "uptime": 4.682  
}
```

## Using Docker Compose

Wouldn't it be great if we could define all the containers we need for our service to work in a simple interface? Welcome to Docker Compose. This is an orchestration tool that enables us to do what we just did in a simpler way and, most importantly, in a distributed way.

We can define how our containers interact with each other and what they need in a simple file structure, and then we can transfer that to another host and just run Docker Compose, where it will deploy it all for us.

Docker Compose uses a YAML file, which is a human-readable data serialization language. It's easier to read and understand and at the same time, allows us to make complex configurations.

Let's migrate our two containers to this layout. The only step we need right now is to create a file called `docker-compose.yml`. Inside, we define our two containers, the network, the ports, and the volumes. Here's an example:

```
version: "3"  
networks:  
  imagini:  
services:  
  database:  
    image: mysql:5.7  
    networks:
```

```
- imagini
volumes:
- ${PWD}/mysql:/var/lib/mysql
environment:
    MYSQL_DATABASE: imagini
    MYSQL_ROOT_PASSWORD: secret
service:
    image: imagini:0.0.5
    networks:
    - imagini
    volumes:
    - ${PWD}/settings.json:/opt/app/settings.json
    ports:
    - "80:3000"
    restart: on-failure
```

Let's do this step by step:

```
version: "3"
```

First, we indicate that we're defining our services in Docker Compose version 3 syntax and features:

```
services:
database:
    image: mysql:5.7
    networks:
    - imagini
    volumes:
    - ${PWD}/mysql:/var/lib/mysql
    environment:
        MYSQL_DATABASE: imagini
        MYSQL_ROOT_PASSWORD: secret
```

Then, we define our database using image `mysql:5.7`, attach it to the `imagini` network, and then use the host `mysql` folder to store the database and define the two environment variables. This is exactly what we did in the previous command:

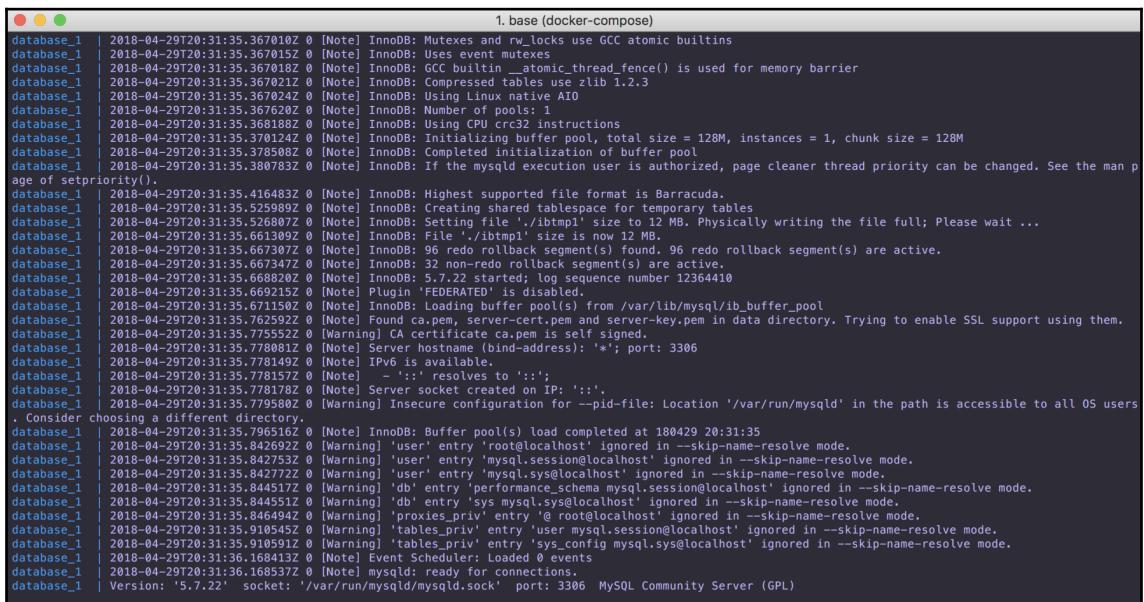
```
service:
    image: imagini:0.0.5
    networks:
    - imagini
    volumes:
    - ${PWD}/settings.json:/opt/app/settings.json
    ports:
    - "80:3000"
    restart: on-failure
```

Finally, we do the same for our service, indicating the name, the `imagini:0.0.5` image, the network, the local `settings` file, and the port assignment. But there's something more, which is very important, and this is the restart policy. Because our database service does not load instantly, our service will most probably fail at first and stop. Docker will then lift it up again and then the database server will be ready, and everything will be up and running.

As we're composing a Docker Compose project, we can avoid using `imagini` on the service names, and so we need to once again change our `settings` file to something like this:

```
{
    "db": "mysql://root:secret@database/imagini"
}
```

We can now try it out by running `docker-compose up`. This will deploy our containers under the `imagini` project, which is the folder name:



```
● base (docker-compose)
[database_1] | 2018-04-29T20:31:35.367010Z 0 [Note] InnoDB: Mutexes and rw_locks use GCC atomic builtins
[database_1] | 2018-04-29T20:31:35.367015Z 0 [Note] InnoDB: Uses event mutexes
[database_1] | 2018-04-29T20:31:35.367018Z 0 [Note] InnoDB: GCC builtin __atomic_thread_fence() is used for memory barrier
[database_1] | 2018-04-29T20:31:35.367021Z 0 [Note] InnoDB: Compressed tables use zlib 1.2.3
[database_1] | 2018-04-29T20:31:35.367024Z 0 [Note] InnoDB: Using Linux native AIO
[database_1] | 2018-04-29T20:31:35.367027Z 0 [Note] InnoDB: Number of pools: 1
[database_1] | 2018-04-29T20:31:35.368188Z 0 [Note] InnoDB: Using CPU CRC32 instructions
[database_1] | 2018-04-29T20:31:35.370124Z 0 [Note] InnoDB: Initializing buffer pool, total size = 128M, instances = 1, chunk size = 128M
[database_1] | 2018-04-29T20:31:35.378508Z 0 [Note] InnoDB: Completed initialization of buffer pool
[database_1] | 2018-04-29T20:31:35.380783Z 0 [Note] InnoDB: If the mysqld execution user is authorized, page cleaner thread priority can be changed. See the man page of setpriority().
[database_1] | 2018-04-29T20:31:35.416483Z 0 [Note] InnoDB: Highest supported file format is Barracuda.
[database_1] | 2018-04-29T20:31:35.525989Z 0 [Note] InnoDB: Creating shared tablespace for temporary tables
[database_1] | 2018-04-29T20:31:35.526807Z 0 [Note] InnoDB: Setting file './ibtmp1' size to 12 MB. Physically writing the file full; Please wait ...
[database_1] | 2018-04-29T20:31:35.661389Z 0 [Note] InnoDB: File './ibtmp1' size is now 12 MB.
[database_1] | 2018-04-29T20:31:35.667307Z 0 [Note] InnoDB: 96 redo rollback segment(s) found, 96 redo rollback segment(s) are active.
[database_1] | 2018-04-29T20:31:35.667347Z 0 [Note] InnoDB: 32 non-redo rollback segment(s) are active.
[database_1] | 2018-04-29T20:31:35.668820Z 0 [Note] InnoDB: 5.7.22 started; log sequence number 12364410
[database_1] | 2018-04-29T20:31:35.669215Z 0 [Note] Plugin 'FEDERATED' is disabled.
[database_1] | 2018-04-29T20:31:35.671150Z 0 [Note] InnoDB: Loading buffer pool(s) from /var/lib/mysql/ib_buffer_pool
[database_1] | 2018-04-29T20:31:35.762592Z 0 [Note] Found ca.pem, server-cert.pem and server-key.pem in data directory. Trying to enable SSL support using them.
[database_1] | 2018-04-29T20:31:35.775527Z 0 [Warning] CA certificate ca.pem is self signed.
[database_1] | 2018-04-29T20:31:35.778081Z 0 [Note] Server hostname (bind-address): '*' port: 3306
[database_1] | 2018-04-29T20:31:35.778149Z 0 [Note] IPv6 is available.
[database_1] | 2018-04-29T20:31:35.778157Z 0 [Note] - '::' resolves to '::';
[database_1] | 2018-04-29T20:31:35.778178Z 0 [Note] Server socket created on IP: '::'.
[database_1] | 2018-04-29T20:31:35.779580Z 0 [Warning] Insecure configuration for --pid-file: Location '/var/run/mysqld' in the path is accessible to all OS users
.. Consider choosing a different directory.
[database_1] | 2018-04-29T20:31:35.796516Z 0 [Note] InnoDB: Buffer pool(s) load completed at 180429 20:31:35
[database_1] | 2018-04-29T20:31:35.842692Z 0 [Warning] 'user' entry 'root@localhost' ignored in --skip-name-resolve mode.
[database_1] | 2018-04-29T20:31:35.842753Z 0 [Warning] 'user' entry 'mysql.session@localhost' ignored in --skip-name-resolve mode.
[database_1] | 2018-04-29T20:31:35.842772Z 0 [Warning] 'user' entry 'mysql.sys@localhost' ignored in --skip-name-resolve mode.
[database_1] | 2018-04-29T20:31:35.844517Z 0 [Warning] 'db' entry 'performance_schema mysql.session@localhost' ignored in --skip-name-resolve mode.
[database_1] | 2018-04-29T20:31:35.844551Z 0 [Warning] 'db' entry 'sys mysql.sys@localhost' ignored in --skip-name-resolve mode.
[database_1] | 2018-04-29T20:31:35.846494Z 0 [Warning] 'proxies_priv' entry '@ root@localhost' ignored in --skip-name-resolve mode.
[database_1] | 2018-04-29T20:31:35.910545Z 0 [Warning] 'tables_priv' entry 'user mysql.session@localhost' ignored in --skip-name-resolve mode.
[database_1] | 2018-04-29T20:31:35.910591Z 0 [Warning] 'tables_priv' entry 'sys_config mysql.sys@localhost' ignored in --skip-name-resolve mode.
[database_1] | 2018-04-29T20:31:36.168413Z 0 [Note] Event Scheduler: Loaded 0 events
[database_1] | 2018-04-29T20:31:36.168537Z 0 [Note] mysqld: ready for connections.
[database_1] | Version: '5.7.22' socket: '/var/run/mysqld/mysqld.sock' port: 3306 MySQL Community Server (GPL)
```

This will start the containers, but you'll notice that they didn't start detached. To do that, hit `Ctrl + C` to stop the containers:

```
1. base (bash)
database_1 | 2018-04-29T20:31:35.367024Z 0 [Note] InnoDB: Using Linux native AIO
database_1 | 2018-04-29T20:31:35.367620Z 0 [Note] InnoDB: Number of pools: 1
database_1 | 2018-04-29T20:31:35.368188Z 0 [Note] InnoDB: Using CPU crc32 instructions
database_1 | 2018-04-29T20:31:35.370124Z 0 [Note] InnoDB: Initializing buffer pool, total size = 128M, instances = 1, chunk size = 128M
database_1 | 2018-04-29T20:31:35.378508Z 0 [Note] InnoDB: Completed initialization of buffer pool
database_1 | 2018-04-29T20:31:35.380783Z 0 [Note] InnoDB: If the mysqld execution user is authorized, page cleaner thread priority can be changed. See the man page of setpriority().
database_1 | 2018-04-29T20:31:35.416483Z 0 [Note] InnoDB: Highest supported file format is Barracuda.
database_1 | 2018-04-29T20:31:35.525989Z 0 [Note] InnoDB: Creating shared tablespace for temporary tables
database_1 | 2018-04-29T20:31:35.526807Z 0 [Note] InnoDB: Setting file './ibtmp1' size to 12 MB. Physically writing the file full; Please wait ...
database_1 | 2018-04-29T20:31:35.661309Z 0 [Note] InnoDB: File './ibtmp1' size is now 12 MB.
database_1 | 2018-04-29T20:31:35.667307Z 0 [Note] InnoDB: 98 redo rollback segment(s) found. 96 redo rollback segment(s) are active.
database_1 | 2018-04-29T20:31:35.667347Z 0 [Note] InnoDB: 32 non-redo rollback segment(s) are active.
database_1 | 2018-04-29T20:31:35.668820Z 0 [Note] InnoDB: 5.7.22 started; log sequence number 12364410
database_1 | 2018-04-29T20:31:35.669215Z 0 [Note] Plugin 'FEDERATED' is disabled.
database_1 | 2018-04-29T20:31:35.671158Z 0 [Note] InnoDB: Loading buffer pool(s) from /var/lib/mysql/ib_buffer_pool
database_1 | 2018-04-29T20:31:35.762592Z 0 [Note] Found ca.pem, server-cert.pem and server-key.pem in data directory. Trying to enable SSL support using them.
database_1 | 2018-04-29T20:31:35.775552Z 0 [Warning] CA certificate ca.pem is self signed.
database_1 | 2018-04-29T20:31:35.778801Z 0 [Note] Server hostname (bind-address): '*' port: 3306
database_1 | 2018-04-29T20:31:35.778149Z 0 [Note] IPv6 is available.
database_1 | 2018-04-29T20:31:35.778157Z 0 [Note] - '::' resolves to '::';
database_1 | 2018-04-29T20:31:35.778178Z 0 [Note] Server socket created on IP: '::'.
database_1 | 2018-04-29T20:31:35.779580Z 0 [Warning] Insecure configuration for --pid-file: Location '/var/run/mysqld' in the path is accessible to all OS users
. Consider choosing a different directory.
database_1 | 2018-04-29T20:31:35.796516Z 0 [Note] InnoDB: Buffer pool(s) load completed at 180429 20:31:35
database_1 | 2018-04-29T20:31:35.842692Z 0 [Warning] 'user' entry 'root@localhost' ignored in --skip-name-resolve mode.
database_1 | 2018-04-29T20:31:35.842753Z 0 [Warning] 'user' entry 'mysql.session@localhost' ignored in --skip-name-resolve mode.
database_1 | 2018-04-29T20:31:35.842772Z 0 [Warning] 'user' entry 'mysql.sys@localhost' ignored in --skip-name-resolve mode.
database_1 | 2018-04-29T20:31:35.844517Z 0 [Warning] 'db' entry 'performance_schema.mysql.session@localhost' ignored in --skip-name-resolve mode.
database_1 | 2018-04-29T20:31:35.844551Z 0 [Warning] 'db' entry 'sys.mysql.sys@localhost' ignored in --skip-name-resolve mode.
database_1 | 2018-04-29T20:31:35.846494Z 0 [Warning] 'proxies_priv' entry '@ root@localhost' ignored in --skip-name-resolve mode.
database_1 | 2018-04-29T20:31:35.910545Z 0 [Warning] 'tables_priv' entry 'user.mysql.session@localhost' ignored in --skip-name-resolve mode.
database_1 | 2018-04-29T20:31:35.910591Z 0 [Warning] 'tables_priv' entry 'sys_config.mysql.sys@localhost' ignored in --skip-name-resolve mode.
database_1 | 2018-04-29T20:31:36.168413Z 0 [Note] Event Scheduler: Loaded 0 events
database_1 | 2018-04-29T20:31:36.168537Z 0 [Note] mysqld: ready for connections.
database_1 | Version: '5.7.22' socket: '/var/run/mysqld/mysqld.sock' port: 3306 MySQL Community Server (GPL)
Killing imagini_service_1 ... done
Killing imagini_database_1 ... done
> docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS              NAMES
> -

```

Now, let's run this again properly by using the `-d` parameter:

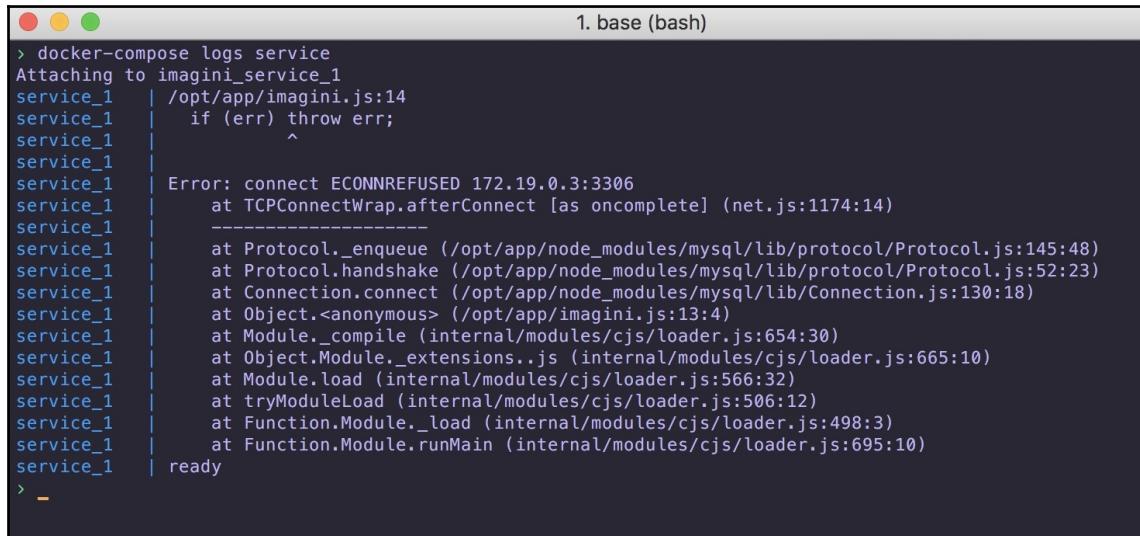
```
1. base (bash)
> docker-compose up -d
Creating network "imagini_imagini" with the default driver
Creating imagini_database_1 ... done
Creating imagini_service_1 ... done
> docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS              NAMES
3318727400a2        mysql:5.7          "docker-entrypoint.s..."   Less than a second ago   Up 5 seconds      3306/tcp           imagini_database_1
bba49f7899f9        imagini:0.0.5     "node /opt/app/imagi..."   Less than a second ago   Up 4 seconds      0.0.0.0:80->3000/tcp   imagini_service_1
> -
```

Great! We can start our service with everything it needs in a single command. You could just share this folder with a co-worker and he or she could have the service up and running in a matter of seconds.

## Mastering Docker Compose

Now that we have our services up and running, we need some more information to manage these services together, and more specifically, to control and monitor its state.

To look at a specific service, you just need its original name. In our case, there's only the service and database to choose from. Let's look at our service:



The screenshot shows a terminal window titled "1. base (bash)". The command entered is "docker-compose logs service". The output shows logs from the "service\_1" container. It starts with an error message: "Error: connect ECONNREFUSED 172.19.0.3:3306" at line 14 of "/opt/app/imagini.js". This is followed by a stack trace for the MySQL protocol module. Finally, the word "ready" is printed, indicating the service has started.

```
> docker-compose logs service
Attaching to imagini_service_1
service_1  | /opt/app/imagini.js:14
service_1  |   if (err) throw err;
service_1  |   ^
service_1  |
service_1  | Error: connect ECONNREFUSED 172.19.0.3:3306
service_1  |     at TCPConnectWrap.afterConnect [as oncomplete] (net.js:1174:14)
service_1  | -----
service_1  |     at Protocol._enqueue (/opt/app/node_modules/mysql/lib/protocol/Protocol.js:145:48)
service_1  |     at Protocol.handshake (/opt/app/node_modules/mysql/lib/protocol/Protocol.js:52:23)
service_1  |     at Connection.connect (/opt/app/node_modules/mysql/lib/Connection.js:130:18)
service_1  |     at Object.<anonymous> (/opt/app/imagini.js:13:4)
service_1  |     at Module._compile (internal/modules/cjs/loader.js:654:30)
service_1  |     at Object.Module._extensions..js (internal/modules/cjs/loader.js:665:10)
service_1  |     at Module.load (internal/modules/cjs/loader.js:566:32)
service_1  |     at tryModuleLoad (internal/modules/cjs/loader.js:506:12)
service_1  |     at Function.Module._load (internal/modules/cjs/loader.js:498:3)
service_1  |     at Function.Module.runMain (internal/modules/cjs/loader.js:695:10)
service_1  | ready
> -
```

As I told you previously, the service will probably fail at first and then Docker will restart it before it finally becomes ready. We can see that in this log. If you want to see the logs of all of the services mixed together by date, you can just omit the service name.

Another important command to memorize is used to list the services that are running. This is the same as for Docker, but it only lists the ones from our configuration:

```
1. base (bash)
> docker-compose ps
      Name           Command       State    Ports
----->
imagini_database_1   docker-entrypoint.sh mysqld   Up      3306/tcp
imagini_service_1    node /opt/app/imagini        Up      0.0.0.0:80->3000/tcp
> -
```

Finally, to stop and remove everything from our deployment, we can run `docker-compose down`. This will first stop the containers and then remove them completely. This ensures that we don't leave any garbage behind:

```
1. base (bash)
> docker-compose ps
      Name           Command       State    Ports
----->
imagini_database_1   docker-entrypoint.sh mysqld   Up      3306/tcp
imagini_service_1    node /opt/app/imagini        Up      0.0.0.0:80->3000/tcp
> docker-compose down
Stopping imagini_database_1 ... done
Stopping imagini_service_1 ... done
Removing imagini_database_1 ... done
Removing imagini_service_1 ... done
Removing network imagini_imagini
> -
```

## Summary

We now have a tool to deploy our service in a consistent way across hosts. We're now able to orchestrate other services, like we did with the database server, in order to have as many dependencies as possible in a controllable state. This ensures that the transition from development to deployment is as smooth as possible.

Let's follow this path and see what else Docker and other tools can do to help us deploy our service. In the following chapter, we'll see how we can take advantage of Docker tools to scale our service by creating replicas. We will see how to distribute our service parts and monitor its state.

# 9

# Scaling, Sharding, and Replicating

We're now able to deploy our microservice almost anywhere with minimal effort. Let's take a look at how we can leverage this and scale our microservice to handle an intensive usage environment.

Before starting, let's see what each of the topics covered in this chapter mean. We'll start from the end. Replicating is the easiest one, and it means to replicate, or copy, your service. Basically, replicating a microservice means it is running multiple instances at the same time, usually on different locations.

Sharding is similar, but with a different purpose. When replicating, each replica can do the full service job. When sharing, each shard can do only part of the service and you need all shards to have your service online. This is a common practice on very large database servers.

Scaling is a common meaning to both replicating and sharding, as both of them allow you to scale your service. Scaling is the process of growing your microservice to handle more load or failure events.

Being able to deploy consistently is important. This gives you better confidence when developing and testing because you have a consistent base layout for your service to run on. Not just that, it also allows you to deploy to multiple locations faster.

This enables you to replicate your service across multiple locations. Replicating a microservice not only allows you to develop in parallel, enabling every developer to have an instance running on its own computer, but it also gives you several advantages in a production environment, such as:

- **Distribution:** When your service is spread across geographic locations, being closer to every customer, reducing latency from every location
- **Fault tolerance:** When your service has an outage or usage peak on specific instances and you're able to route customers to instances being less used
- **Zero downtime:** When your service has enough replicas that even if a substantial part of your infrastructure is affected by an external incident, your service is still generally available

Distributing your service geographically puts it near your customers. Usually, you want to distribute it in different continents to avoid inter-continental latency. If your service is broadly used, you may need to have several instances per continent, perhaps one in every country.

Fault tolerance and zero downtime are related to each other. Being able to operate when instances of your service are faulty gives users the perception of no downtime. This is also very important when you want to make an upgrade to your service without bringing all the instances down. You may phase-in each instance to upgrade while routing your customers to other instances, keeping your global service online in a virtual sense.

In this chapter, we'll see how we can use Docker tools to replicate our service using Swarm. Later on, we'll see how easy it is to migrate our microservice to Kubernetes locally.

## Scaling your network

Scaling a service by itself is not sufficient. Just because you create instances of your service doesn't mean they'll just work together. There are two important steps to enable a service scale:

- Making instances work together
- Making instances reachable

Instances don't need to communicate with each other, but an instance must be able to work without interfering with the others. An instance should be able to handle a request from a client that was previously served by another instance. In our case, this means an instance must be able to manipulate an image that was uploaded by another instance. This ensures that if an instance goes offline, others can take over its clients.

Instances need to be reachable, which means you need to be able to point your customers to the nearest instances that are live and operating normally. This means you need to be able to monitor your instances and distribute your customers to the online instances.

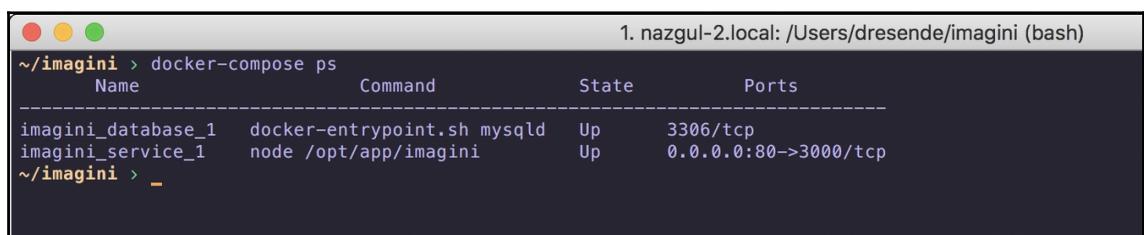
Services are usually reachable by a DNS address. You need to be able to control address name resolution, preferably by geography, to point your customers to the best instances. You can point directly to instances or have proxies monitor a couple of nearby instances and proxy the traffic based on policies.

There are many different kinds of policies. For example, if the proxy is able to monitor how busy an instance is, it can direct new requests to the less busy instances. Or, it can just distribute requests using a round-robin approach.

There's also another common policy that routes traffic according to the IP address. For example, if you have three instances, you use the remainder of the remainder of the devision of the IP address (as a 4-byte integer) by three to choose the instance (possible values would be 0, 1, and 2).

## Replicating our microservice

Picking up from the previous chapter, we ended with a Docker Compose configuration that starts our two containers, one with our microservice and the other with the database server that stores our data:



```
1. nazgul-2.local: /Users/dresende/imagin (bash)
~/imagin > docker-compose ps
      Name           Command       State    Ports
----- 
imagin_database_1   docker-entrypoint.sh mysqld   Up      3306/tcp
imagin_service_1    node /opt/app/imagin        Up      0.0.0.0:80->3000/tcp
~/imagin > _
```

You may have noticed when deploying with Docker Compose that our two containers have a `_1` suffix on the name. This is because Docker Compose supports replicating our instance using Docker swarm.

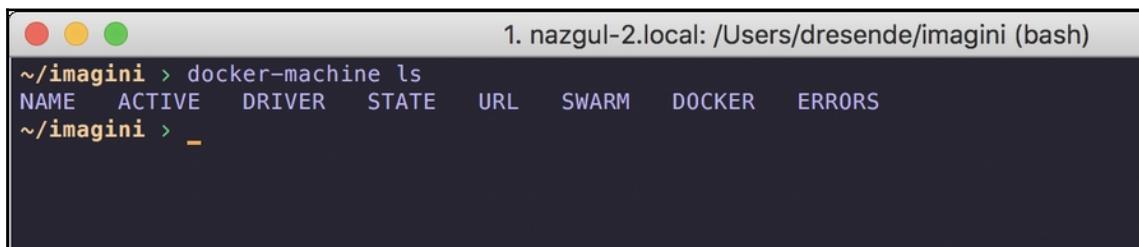
Docker swarm is another component of the Docker engine that enables you to create a cluster of hosts to run your microservice instances while orchestrating all of this using a simple tool from any of the hosts in your cluster.

Swarm is the name Docker uses to reference a cluster of nodes. Let's create a new Swarm using our host as the manager of the swarm. Since I'm using Docker for Mac, I need to create another Docker virtual machine to be able to simulate a couple of hosts. If you're using a Linux machine, you can just test this with two different hosts.

If you're using macOS like me, follow these steps. We'll use Docker machine to create two virtual machines to run as Docker hosts. First, let's see what machines we have created by typing:

```
docker-machine ls
```

You should see an output as follows:



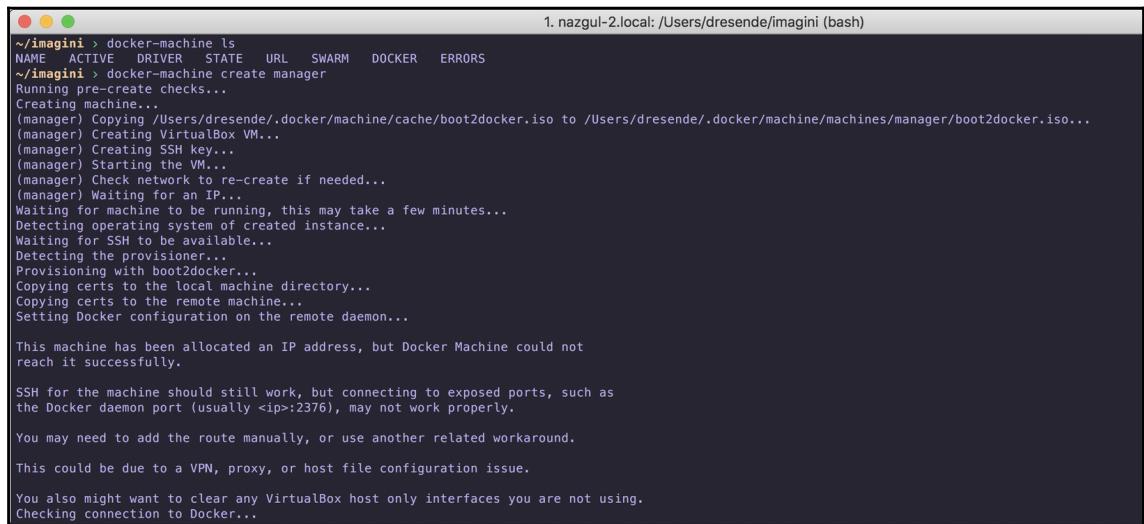
The screenshot shows a macOS terminal window with three colored window control buttons (red, yellow, green) at the top left. The title bar says "1. nazgul-2.local: /Users/dresende/imagini (bash)". The command `~/imagini > docker-machine ls` is entered, followed by a table with the following data:

NAME	ACTIVE	DRIVER	STATE	URL	SWARM	DOCKER	ERRORS
~/imagini	-	-	-	-	-	-	-

No machines. Let's create a manager node. To do that, run:

```
docker-machine create manager
```

Docker will create the new machine, assign a new IP address, and prepare the Docker engine inside the machine:



```
1. nazgul-2.local: /Users/dresende/imagini (bash)
~/imagini > docker-machine ls
NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS
~/imagini > docker-machine create manager
Running pre-create checks...
Creating machine...
(manager) Copying /Users/dresende/.docker/machine/cache/boot2docker.iso to /Users/dresende/.docker/machine/machines/manager/boot2docker.iso...
(manager) Creating VirtualBox VM...
(manager) Creating SSH key...
(manager) Starting the VM...
(manager) Check network to re-create if needed...
(manager) Waiting for an IP...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...

This machine has been allocated an IP address, but Docker Machine could not
reach it successfully.

SSH for the machine should still work, but connecting to exposed ports, such as
the Docker daemon port (usually <ip>:2376), may not work properly.

You may need to add the route manually, or use another related workaround.

This could be due to a VPN, proxy, or host file configuration issue.

You also might want to clear any VirtualBox host only interfaces you are not using.

Checking connection to Docker...
```

Now, run the following command:

```
docker-machine ls
```

You'll see that our new machine is ready to be used. You can see the IP address of the machine in the URL column:



```
1. nazgul-2.local: /Users/dresende/imagini (bash)
~/imagini > docker-machine ls
NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS
manager * virtualbox Running tcp://192.168.99.100:2376 v18.03.1-ce
~/imagini > _
```

The Docker engine has an exposed API that enables you to manage a Docker host using the Docker commands from another host. The URL column indicates the API address to manage that specific host.

We can use a little helper command to configure our Docker command to manage this new host. Just type the following command:

```
docker-machine env manager
```

And follow the instructions:

```
1. nazgul-2.local: /Users/dresende/imagini (bash)
~/imagini > docker-machine ls
NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS
manager * virtualbox Running tcp://192.168.99.100:2376 v18.03.1-ce
~/imagini > docker-machine env manager
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="/Users/dresende/.docker/machine/machines/manager"
export DOCKER_MACHINE_NAME="manager"
# Run this command to configure your shell:
# eval $(docker-machine env manager)
~/imagini > eval $(docker-machine env manager)
~/imagini > docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
~/imagini > _
```

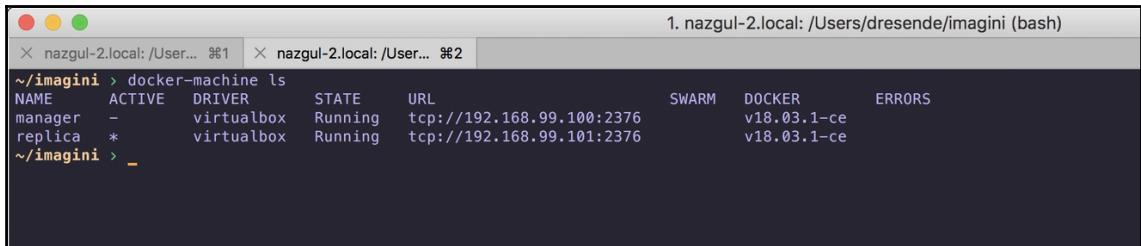
Now, let's keep this Terminal untouched and open a new tab so that we can create a second machine. Let's call it replica, which will hold a second instance of our microservice:

```
1. nazgul-2.local: /Users/dresende/imagini (bash)
X nazgul-2.local: /User... %1 X nazgul-2.local: /User... %2
~/imagini > docker-machine ls
NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS
manager - virtualbox Running tcp://192.168.99.100:2376 v18.03.1-ce
~/imagini > docker-machine create replica
Running pre-create checks...
Creating machine...
(replica) Copying /Users/dresende/.docker/machine/cache/boot2docker.iso to /Users/dresende/.docker/machine/machines/replica/boot2docker.iso...
(replica) Creating VirtualBox VM...
(replica) Creating SSH key...
(replica) Starting the VM...
(replica) Check network to re-create if needed...
(replica) Waiting for an IP...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
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 this virtual machine, run: docker-machine env replica
~/imagini >
~/imagini > _
```

We can check that our new virtual machine is ready and running by checking the following output once more:

```
docker-machine ls
```

Both of our machines should be ready and running:



```
1. nazgul-2.local: /Users/dresende/imagini (bash)
~ nazgul-2.local: /User... %1 | X nazgul-2.local: /User... %2 |
~/imagini > docker-machine ls
NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS
manager - virtualbox Running tcp://192.168.99.100:2376 v18.03.1-ce
replica * virtualbox Running tcp://192.168.99.101:2376 v18.03.1-ce
~/imagini >
```

Using our initial tab, where we have the manager host, let's create a swarm. We're indicating the advertisement address because our virtual machines have more addresses, but we want the advertisement to be done on the network shown previously, so we're pointing out a specific IP address at the end:



```
1. nazgul-2.local: /Users/dresende/imagini (bash)
~ nazgul-2.local: /User... %1 | X nazgul-2.local: /User... %2 |
~/imagini > docker swarm init --advertise-addr 192.168.99.100
Swarm initialized: current node (f5c0x801fxosc7auasl6h0l01) is now a manager.

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

  docker swarm join --token SWMTKN-1-62mxv9trit2gcs24icbsu9cmj4ezg1bzcry768uee5roqgcrcx-76275qfmzi3u3dyrujm25cepb 192.168.99.100:2377

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

Docker created our swarm and gave us instructions on how to make other hosts join the swarm using a kind of secret token. If you want to add more nodes later on, save that command.

Now, head to the second tab and let's change our Docker command to manage our replica host and then join the swarm:

```

1. nazgul-2.local: /Users/dresende/imagini (bash)
~/imagini > docker-machine ls
NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS
manager - virtualbox Running tcp://192.168.99.100:2376 v18.03.1-ce
replica - virtualbox Running tcp://192.168.99.101:2376 v18.03.1-ce
~/imagini > docker-machine env replica
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.101:2376"
export DOCKER_CERT_PATH="/Users/dresende/.docker/machine/machines/replica"
export DOCKER_MACHINE_NAME="replica"
# Run this command to configure your shell:
# eval $(docker-machine env replica)
~/imagini > eval $(docker-machine env replica)
~/imagini > docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
~/imagini > docker swarm join --token SWMTKN-1-62mxv9trit2gcs24icbsu9cmj4ezg1bzcry768uee5roqgcrcx-76275qfmzi3u3dyrujm25cepb 192.168.99.100:2377
This node joined a swarm as a worker.
~/imagini >

```

We now have two nodes on our swarm. From this point forward, assume that the first tab of our console is the manager host and that the second tab is our replica host.

We can check that the swarm has our two active nodes by using the `docker node ls` command from our manager:

```

1. nazgul-2.local: /Users/dresende/imagini (bash)
~/imagini > docker node ls
ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS ENGINE VERSION
f5c0x801fxosc7auasl6h0l0i * manager Ready Active Leader 18.03.1-ce
0po46p8ijuqzoe6g6chg432x replica Ready Active
~/imagini >

```

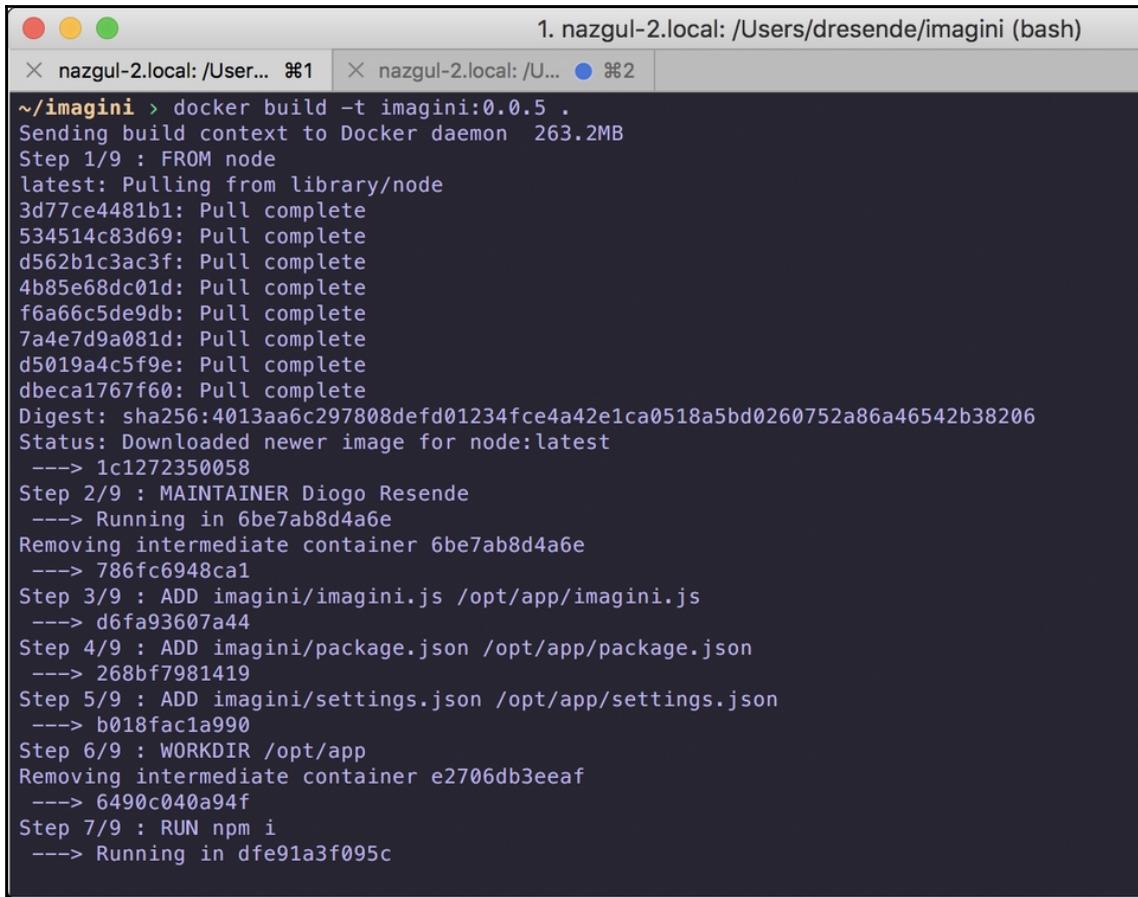
## Deploying to swarm

We have two clean nodes in our swarm. We need to go back a little bit and prepare our hosts for our service. More specifically, we need to create our image again.

Because we have our `Dockerfile`, this is an easy task. Just run:

```
docker build -t imagini:0.0.5 .
```

Do this on both hosts:



The screenshot shows a terminal window titled "1. nazgul-2.local: /Users/dresende/imagini (bash)". It displays the command "docker build -t imagini:0.0.5 ." followed by the Docker build process. The output shows the pulling of the "node" image from the Docker daemon, the creation of intermediate containers for each step, and the final successful build of the "imagini" image with digest sha256:4013aa6c297808defd01234fce4a42e1ca0518a5bd0260752a86a46542b38206.

```
~/imagini > docker build -t imagini:0.0.5 .
Sending build context to Docker daemon 263.2MB
Step 1/9 : FROM node
latest: Pulling from library/node
3d77ce4481b1: Pull complete
534514c83d69: Pull complete
d562b1c3ac3f: Pull complete
4b85e68dc01d: Pull complete
f6a66c5de9db: Pull complete
7a4e7d9a081d: Pull complete
d5019a4c5f9e: Pull complete
dbeaca1767f60: Pull complete
Digest: sha256:4013aa6c297808defd01234fce4a42e1ca0518a5bd0260752a86a46542b38206
Status: Downloaded newer image for node:latest
--> 1c1272350058
Step 2/9 : MAINTAINER Diogo Resende
--> Running in 6be7ab8d4a6e
Removing intermediate container 6be7ab8d4a6e
--> 786fc6948ca1
Step 3/9 : ADD imagini/imagini.js /opt/app/imagini.js
--> d6fa93607a44
Step 4/9 : ADD imagini/package.json /opt/app/package.json
--> 268bf7981419
Step 5/9 : ADD imagini/settings.json /opt/app/settings.json
--> b018fac1a990
Step 6/9 : WORKDIR /opt/app
Removing intermediate container e2706db3eeaf
--> 6490c040a94f
Step 7/9 : RUN npm i
--> Running in dfe91a3f095c
```

Finally, we'll have our image available on both nodes. Our database container also needs an image, but since that's an official, published image, we don't need to build it, as Docker will download it when it needs it.

We can now use the Docker stack tool to deploy our instances. It uses our previous Docker Compose configuration to know how to deploy our service. But before we do this, we need to make a couple of adjustments to the configuration.

For now, we'll enforce our database to only have one replica, as we're not yet prepared to distribute our database server just yet. We can do that by having this section on the configuration:

```
deploy:  
  replicas: 1  
  placement:  
    constraints: [node.role == manager]
```

It indicates that we only want one replica and that the container should be placed (should run) in the manager node.

Another change we need to do is change the database volume because our containers are running on a virtual machine that is no longer local. Let's change the volume section to this:

```
volumes:  
- /var/lib/mysql:/var/lib/mysql
```

We need to create the folder on the manager machine. To do that, run:

```
docker-machine ssh manager 'mkdir /var/lib/mysql'
```

To sum up, you should have a configuration as follows:

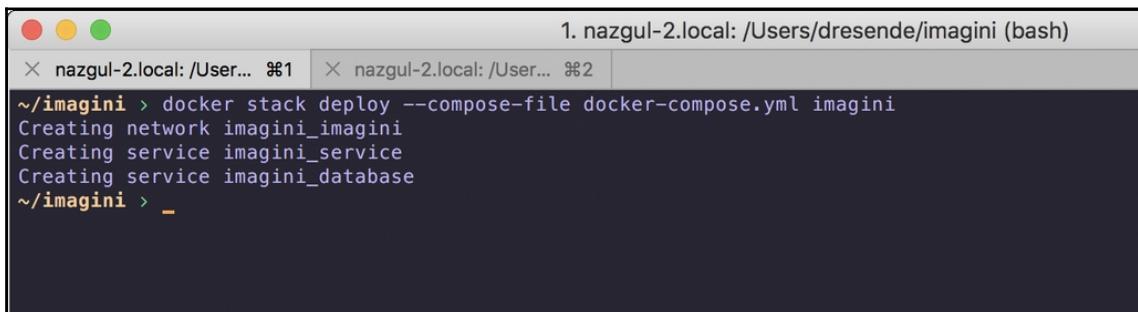
```
version: "3"  
networks:  
  imaginis:  
services:  
  database:  
    image: mysql:5.7  
    networks:  
    - imaginis  
    volumes:  
    - /var/lib/mysql:/var/lib/mysql  
    ports:  
    - "3306:3306"  
    environment:  
      MYSQL_DATABASE: imaginis  
      MYSQL_ROOT_PASSWORD: secret  
    deploy:  
      replicas: 1  
      placement:  
        constraints: [node.role == manager]  
  service:  
    image: imaginis:0.0.5  
    networks:  
    - imaginis  
    volumes:
```

```
- ${PWD}/settings.json:/opt/app/settings.json
ports:
- "80:3000"
```

Head to the first tab, the one that controls the manager node, and run the following:

```
docker stack deploy --compose-file docker-compose.yml imagini
```

This should start the deployment. It will create the network on the swarm and then deploy the two services:



```
1. nazgul-2.local: /Users/dresende/imagini (bash)
~/imagini > docker stack deploy --compose-file docker-compose.yml imagini
Creating network imagini_imagini
Creating service imagini_service
Creating service imagini_database
~/imagini > _
```

Wait a little bit and then run the following command:

```
docker stack ps imagini
```

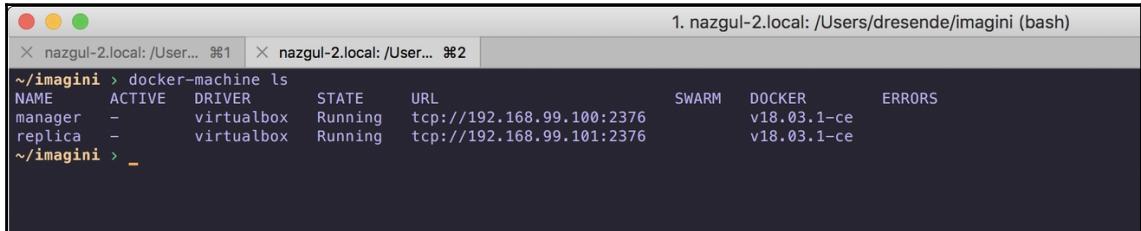
This will do the same as `docker ps`, but just for our stack:



ID	NAME	IMAGE	NODE	DESIRED STATE	CURRENT STATE	ERROR	PORTS
qu1xuij4xngd	imagini_service.1	imagini:0.0.5	replica	Running	Running 9 seconds ago		
k6by9mnzj6k5	imagini_database.1	mysql:5.7	manager	Running	Running 13 seconds ago		

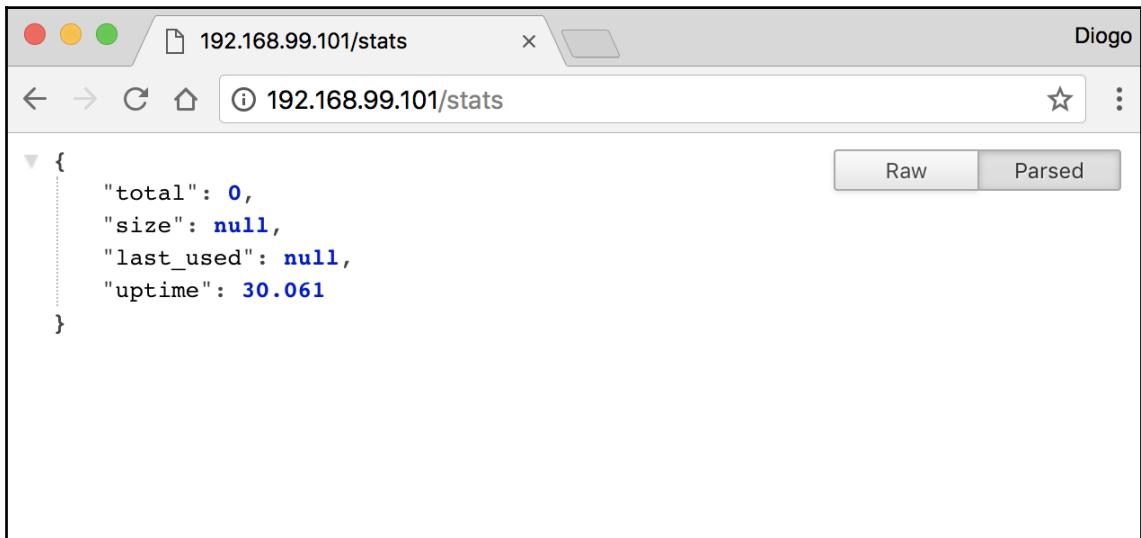
The only difference you may notice is that the name of the containers have a `.1` on the end of them and that a new `NODE` column indicates where it's running inside our swarm.

Because our containers are running in a swarm inside two virtual machines, we need to use the IP address of the node that has the service running on it. Looking at the previous screenshot, we can see that it's in the replica machine:

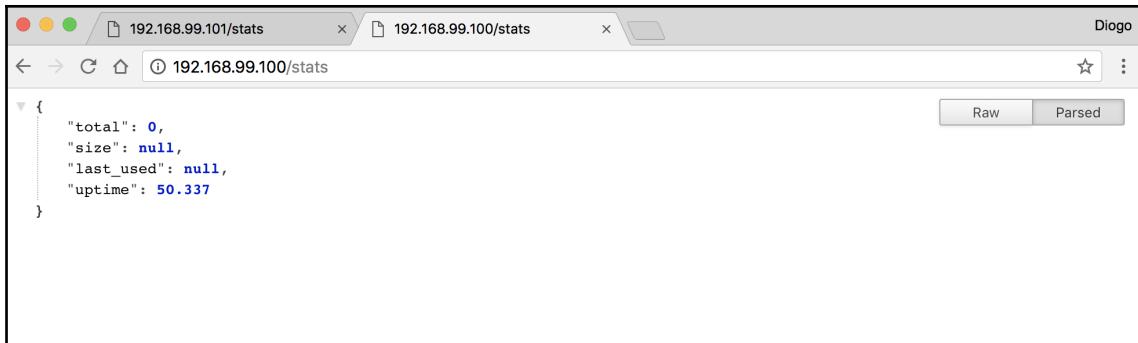


```
1. nazgul-2.local: /Users/dresende/imagini (bash)
x nazgul-2.local: /User... % 1 x nazgul-2.local: /User... % 2
~/imagini > docker-machine ls
NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS
manager - virtualbox Running tcp://192.168.99.100:2376 v18.03.1-ce
replica - virtualbox Running tcp://192.168.99.101:2376 v18.03.1-ce
~/imagini >
```

Its address is 192.168.99.101. Head to the browser and see if our service is available or not:



Great! It's working. But, if you think about it, if this is a swarm, shouldn't it be available from anywhere in the network? You're right, yes, it should be available. Let's check the other node address:



Now, this is something awesome. Notice that we only had one instance of each of our two services, and our `imagini` service was on the replica node. Although it's available from any of the swarm node addresses, if, for some reason, it fails, you wouldn't be able to reach it.

We can change this by scaling the number of instances, or replicas, in our swarm. To do that, just issue the following command to change the scale to two instances:

```
docker service scale imagini_service=2
```

Docker will handle the deployment and check if everything goes as planned:



You can check the status of the services at any time using the following command:

```
docker service ls
```

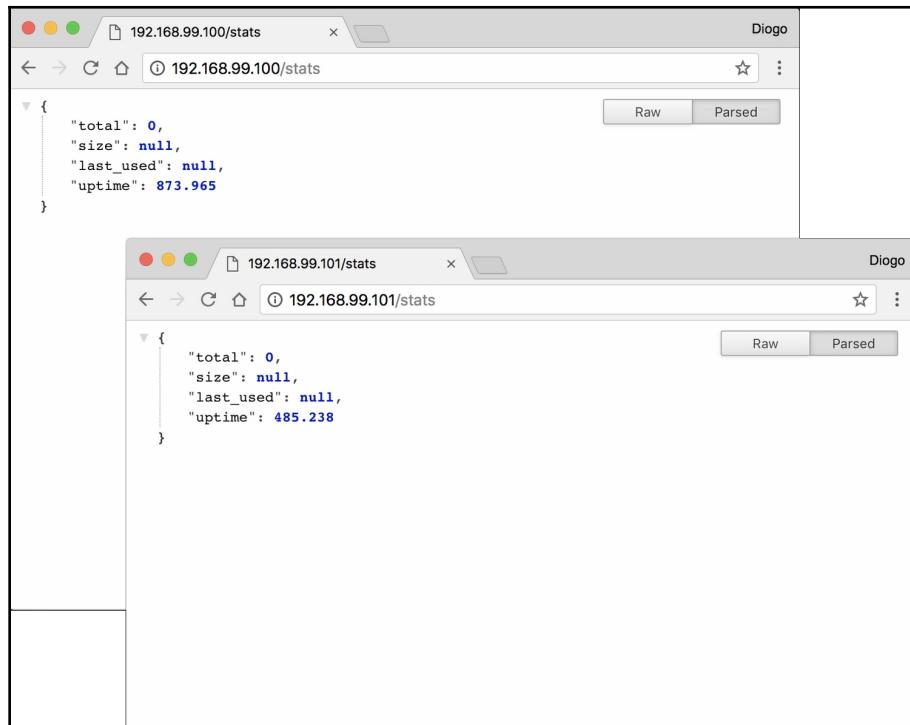
Notice how our service 2/2 replicas are running:



```
× nazgul-2.local: /User... ⌘1 × nazgul-2.local: /User... ⌘2
~/imagini $ docker service ls
ID          NAME      MODE      REPLICAS  IMAGE
ccearve1h836  imaginis_database replicated  1/1      mysql:5.7
a91zf8z4mn6t  imaginis_service   replicated  2/2      imaginis:0.0.5
~/imagini $ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS
12dd8b830fa6      imaginis:0.0.5    "node /opt/app/imagin..." 48 seconds ago   Up 47 seconds       3000/tcp
d4351526772a      mysql:5.7       "docker-entrypoint.s..." 7 minutes ago    Up 7 minutes        3306/tcp
~/imagini : -
```

If you head to the browser, both addresses still work as expected, but in the background, we have two services running. You can notice this by looking at the `uptime` property.

If you run a little test, such as refreshing both addresses repeatedly, you'll notice that the `uptime` changes up and down. This is because we deployed both of the instances at different times, and although the statistics come from the database server, which is the same, the `uptime` is the process uptime:

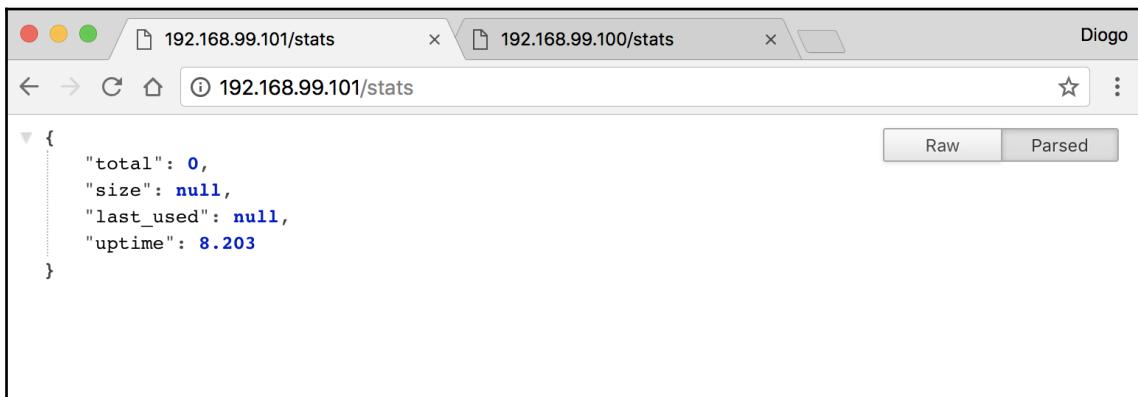


Notice how the swarm is not constant and does not give you the same instance for the same address. It keeps rotating it.

The swarm also monitors our container instances. For example, let's imagine you're working on a host and, by accident, stop a container:

```
1. nazgul-2.local: /Users/dresende/imagini (bash)
~/imagini > docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                 NAMES
12dd8b83af6          imagini:0.0.5      "node /opt/app/imagi..."   14 minutes ago     Up 14 minutes       3000/tcp, 3306/tcp   imagini_service.2.tpmgp03vgip761gwuj3x0zpx6
d4351526772a          mysql:5.7         "docker-entrypoint.s..."  21 minutes ago     Up 21 minutes       3306/tcp           imagini_database.1.rf24m832ek4nstio6718axroy
~/imagini > docker stop 12dd8b83af6
12dd8b83af6
~/imagini >
```

It's not very critical since we have two replicas running. But if you head to the browser and hit refresh, you may see something odd:



What happened to our `uptime`? It just went down to a few seconds. This is because the swarm just noticed our container stopping and restarted it. If you look at the Docker containers again, it's still running. Well, actually it restarted:

```
1. nazgul-2.local: /Users/dresende/imagini (bash)
~/imagini > docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                 NAMES
12dd8b83af6          imagini:0.0.5      "node /opt/app/imagi..."   14 minutes ago     Up 14 minutes       3000/tcp, 3306/tcp   imagini_service.2.tpmgp03vgip761gwuj3x0zpx6
d4351526772a          mysql:5.7         "docker-entrypoint.s..."  21 minutes ago     Up 21 minutes       3306/tcp           imagini_database.1.rf24m832ek4nstio6718axroy
0cef08a5127          imagini:0.0.5      "node /opt/app/imagi..."   16 seconds ago    Up 14 seconds       3000/tcp, 3306/tcp   imagini_service.2.tda7rmkyeyxl7ep96vcc504e
~/imagini > docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                 NAMES
0cef08a5127          imagini:0.0.5      "node /opt/app/imagi..."   16 seconds ago    Up 14 seconds       3000/tcp, 3306/tcp   imagini_service.2.tda7rmkyeyxl7ep96vcc504e
d4351526772a          mysql:5.7         "docker-entrypoint.s..."  21 minutes ago     Up 21 minutes       3306/tcp           imagini_database.1.rf24m832ek4nstio6718axroy
~/imagini >
```

If you want to scale to more instances, you don't need to have more swarm nodes. There's no limitation on the number of instances of the same containers running on the same node. This is actually a good practice.

It enables you to test if your service is ready to be used in a scaled environment and also allows you to test a phased upgrade by stopping a container and upgrading one by one, while at least one other container is still serving your customers.

Let's just scale our service to five instances:



```

nazgul-2.local: /User... %1 nazgul-2.local: /User... %2
~/imagini > docker service scale imaginini_service=5
imaginini_service scaled to 5
overall progress: 5 out of 5 tasks
1/5: running [=====>]
2/5: running [=====>]
3/5: running [=====>]
4/5: running [=====>]
5/5: running [=====>]
verify: Service converged
~/imagini > docker service ls
ID           NAME      MODE      REPLICAS   IMAGE
ccarye1hb36  imaginini_database  replicated  1/1      mysql:5.7
a912f8z4mm6t  imaginini_service  replicated  5/5      imaginini:0.0.5
~/imagini > docker ps
CONTAINER ID   IMAGE      COMMAND     CREATED    STATUS    PORTS          NAMES
c4b7bd8ea132  imaginini:0.0.5  "node /opt/app/imagin..."  21 seconds ago Up 21 seconds  3000/tcp       imaginini_service.4.09kiecrdin7qa485wr1g3xxa
0cef08a5127   imaginini:0.0.5  "node /opt/app/imagin..."  10 minutes ago Up 10 minutes  3000/tcp       imaginini_service.2.lda7rmkyevxl7egp96vccc504e
d4351526772a  mysql:5.7      "docker-entrypoint.s..."  32 minutes ago Up 32 minutes  3306/tcp       imaginini_database.1.rf24m832ek4nsti96718axroy
~/imagini >

```

Notice how we have two of the five services running on this node, and also the database instance. The other three are in the replica node. This is known as swarm balancing the instances.

## Creating services

You may have noticed how we're using the Docker service command to scale our instances. This is a command that is similar to the basic Docker command, but with services and scaling in mind. We can use this command to create containers and scale them easily.

MySQL has no simple replication mechanism. There are only two ways of doing it and both have disadvantages:

- Creating a set of nodes as masters of each other, creating a complex mesh of connections that will eventually lead to chaos
- Creating a cluster, which involves more nodes and still leads to a complex deployment, which is easier to manage but resource-intensive to maintain

Looking back at our examples from previous chapters, we could change our service to use another database server. More specifically, RethinkDB has a much friendlier cluster mechanism.

So, just replace our `imagini` service with the one we used with RethinkDB. This time, let's use the Docker Service commands manually and deploy our microservice step by step.

To begin with, let's prepare a RethinkDB cluster. To make it resilient, we will do the following:

1. Create an instance called `db-primary`.
2. Create another instance called `db-secondary` and instruct it to join `db-primary`.
3. Scale `db-secondary` to two instances so that we have a proper three-node cluster.
4. Remove and recreate `db-primary` to instruct it to join `db-secondary` this time.
5. Scale `db-primary` to two instances.

This will give you four nodes, two of each kind, that are formatted to join the other kind when they fail, and swarm restarts them. This is the definition of *no single point of failure*.

Because we're doing everything manually, we need to start by creating a network. Let's create the network `imagini`:

```
docker network create --driver overlay imagini
```

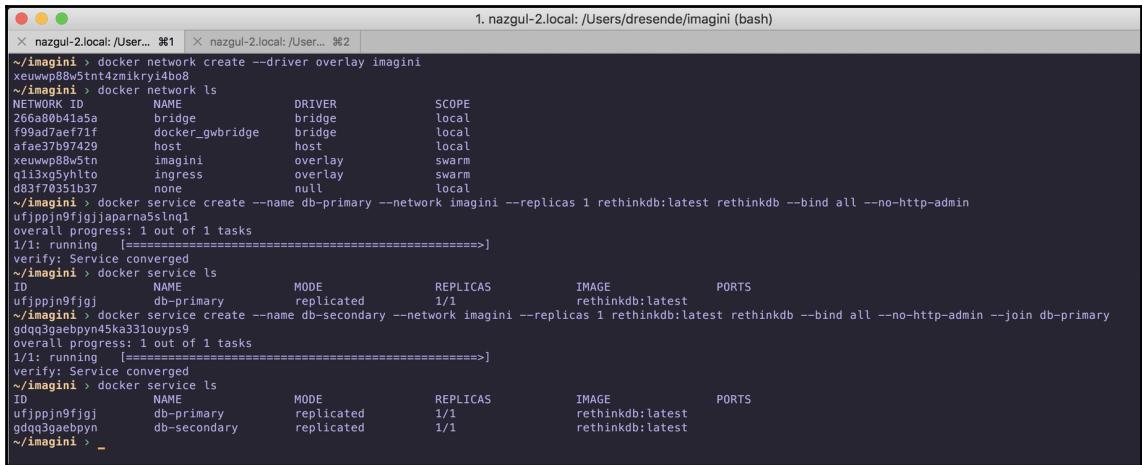
Then, create `db-primary` with only 1 replica as we're removing it later on:

```
docker service create --name db-primary \
    --network imagini \
    --replicas 1 \
    rethinkdb:latest \
    rethinkdb --bind all --no-http-admin
```

Then, create the first `db-secondary` with only 1 replica and tell it to join `db-primary`:

```
docker service create --name db-secondary \
    --network imagini \
    --replicas 1 \
    rethinkdb:latest \
    rethinkdb --bind all --no-http-admin --join db-
primary
```

You should now have something like this:



The screenshot shows a terminal window with two tabs open. The active tab is titled 'nazgul-2.local: /Users/dresende/imagini (bash)'. The command history and output are as follows:

```
~/nazgul-2.local: /User... %1 nazgul-2.local: /User... %2
~/imagini > docker network create --driver overlay imagini
xeuwwp88w5tn4zmikryi4bo8
~/imagini > docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
266a80b41a5a   bridge    bridge      local
f99ad7ae71f    docker_gwbridge  bridge      local
afae37b97429   host      host       local
xeuwwp88w5tn   imagini   overlay     swarm
ql13xq5yhito  ingress   overlay     swarm
d83t70351b37   none      null       local
~/imagini > docker service create --name db-primary --network imagini --replicas 1 rethinkdb:latest rethinkdb --bind all --no-http-admin
utjppjn9fjgjlaparna5tnd1
overall progress: 1 out of 1 tasks
1/1: running  [=====]
verify: Service converged
~/imagini > docker service ls
ID          NAME      MODE      REPLICAS      IMAGE      PORTS
utjppjn9fjgj db-primary replicated 1/1        rethinkdb:latest
~/imagini > docker service create --name db-secondary --network imagini --replicas 1 rethinkdb:latest rethinkdb --bind all --no-http-admin --join db-primary
g0qq3gaebpyn45ka33louyps9
overall progress: 1 out of 1 tasks
1/1: running  [=====]
verify: Service converged
~/imagini > docker service ls
ID          NAME      MODE      REPLICAS      IMAGE      PORTS
utjppjn9fjgj db-primary replicated 1/1        rethinkdb:latest
g0qq3gaebpyn db-secondary replicated 1/1        rethinkdb:latest
~/imagini >
```

Similar to what we did before, let's scale db-secondary to two instances:

```
docker service scale db-secondary=2
```

Then, remove the db-primary and recreate it:

```
docker service create --name db-primary \
--network imagini \
--replicas 1 \
rethinkdb:latest \
rethinkdb --bind all --no-http-admin --join db-
secondary
```

Then, scale it to two instances:

```
docker service scale db-primary=2
```

You should now have four instances. List the services and you should see something like the following screenshot:

```
1. nazgul-2.local: /Users/dresende/imagini (bash)
~/.nazgul-2.local: /User... %1 | ~/.nazgul-2.local: /User... %2 |
~/imagini > docker service ls
ID           NAME      MODE      REPLICAS      IMAGE
siap9vo0bw8b db-primary replicated  2/2          rethinkdb:latest
gdqq3gaebpyn db-secondary replicated 2/2          rethinkdb:latest
~/imagini > -
```

If you try to stop the containers, you'll notice that they will restart without you doing anything. Just give it a little time and they'll be up and running again:

```
1. nazgul-2.local: /Users/dresende/imagini (bash)
~/.nazgul-2.local: /User... %1 | ~/.nazgul-2.local: /User... %2 |
~/imagini > docker ps
CONTAINER ID   IMAGE      COMMAND      CREATED     STATUS      PORTS     NAMES
3604a3ca3e8c   rethinkdb:latest "rethinkdb --bind al..." 39 seconds ago Up 33 seconds  8080/tcp, 28015/tcp, 29015/tcp db-secondary.2.ivznyelargxtzeui2k2crolya
57340d3e2aa   rethinkdb:latest "rethinkdb --bind al..." 4 minutes ago Up 4 minutes  8080/tcp, 28015/tcp, 29015/tcp db-primary.1.0j7lek8ventlz2u34i4e6prcl
~/imagini > docker stop 3604a3ca3e8c 57340d3e2aa
3604a3ca3e8c
57340d3e2aa
~/imagini > docker ps
CONTAINER ID   IMAGE      COMMAND      CREATED     STATUS      PORTS     NAMES
~/imagini > docker ps
CONTAINER ID   IMAGE      COMMAND      CREATED     STATUS      PORTS     NAMES
b865c0a6e719   rethinkdb:latest "rethinkdb --bind al..." 10 seconds ago Up 4 seconds  8080/tcp, 28015/tcp, 29015/tcp db-primary.1.9s2pt59wofavucliy86cau8ss
ed6ab63ed52   rethinkdb:latest "rethinkdb --bind al..." 10 seconds ago Up 4 seconds  8080/tcp, 28015/tcp, 29015/tcp db-secondary.2.0mj2itoz7l3eqkrzrc0t4eg5
~/imagini > -
```

Now that we have our database cluster ready, we just need a proxy to be able to access the cluster:

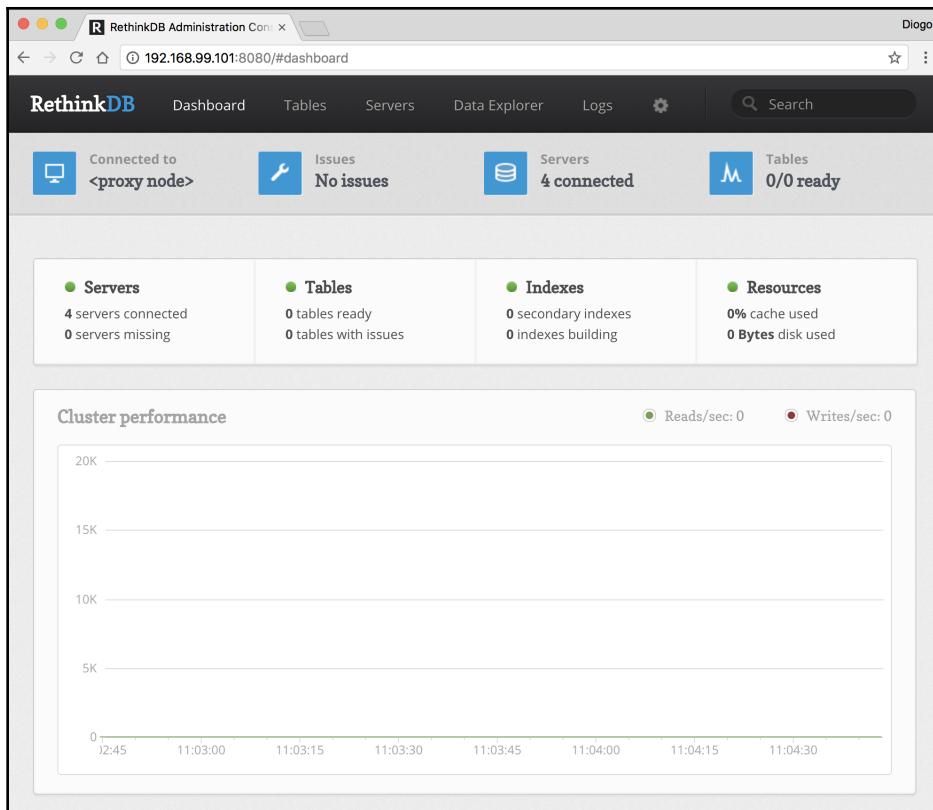
```
docker service create --name database \
--network imagini \
--publish 8080:8080 \
--publish 28015:28015 \
rethinkdb:latest \
rethinkdb proxy --bind all --join db-primary
```

If we look at the services, we can now see that we have our four cluster nodes and a proxy called database, which we'll use to manage and access the data:



```
1. nazgul-2.local: /Users/dresende/imagini (bash)
nazgul-2.local: /User... %1 nazgul-2.local: /User... %2
~/imagini $ docker service create --name database --network imagini --publish 8080:8080 --publish 28015:28015 rethinkdb:latest rethinkdb proxy --bind all --join db-primary
ljqx8icha/x7v9owf5v1bm2h9
overall progress: 1 out of 1 tasks
[1/1] [====>] 100%
verify: service converged
~/imagini $ docker service ls
ID          NAME      MODE      REPLICAS  IMAGE
ljqx8icha/x7v9owf5v1bm2h9   database  replicated  1/1      rethinkdb:latest
siap9v0obwBb    db-primary  replicated  2/2      rethinkdb:latest
gdqq3gaebpyn  db-secondary  replicated  2/2      rethinkdb:latest
~/imagini $
```

We can also see that we have the management port exposed, 8080. Head to the browser at any of the swarm addresses on that port and see if you can see the RethinkDB management interface:



The screenshot shows the RethinkDB Administration Console dashboard. At the top, it displays the URL `192.168.99.101:8080/#dashboard`. Below the header, there are four main status indicators:

- Connected to <proxy node>**: Shows a green icon and text indicating 4 servers connected and 0 servers missing.
- Issues**: Shows a green icon and text indicating No issues.
- Servers**: Shows a green icon and text indicating 4 connected.
- Tables**: Shows a green icon and text indicating 0/0 ready.

Below these, there are four cards with performance metrics:

- Servers**: 4 servers connected, 0 servers missing.
- Tables**: 0 tables ready, 0 tables with issues.
- Indexes**: 0 secondary indexes, 0 indexes building.
- Resources**: 0% cache used, 0 Bytes disk used.

At the bottom, there is a section titled "Cluster performance" with a line graph showing reads/sec and writes/sec over time. The graph has a Y-axis from 0 to 20K and an X-axis from 11:03:45 to 11:04:30. The legend indicates 0 reads/sec and 0 writes/sec.

Great! We can see that there's a **4 servers connected** under the **Servers** label. On top, there's also a **Servers** section that you can look into and see that our instances are all there:

The screenshot shows the RethinkDB Administration Console interface. At the top, there are navigation links: Dashboard, Tables, Servers, Data Explorer, Logs, and a search bar. Below the navigation, there are four status indicators: "Connected to <proxy node>" (blue icon), "Issues No issues" (blue icon), "Servers 4 connected" (blue icon), and "Tables 0/0 ready" (blue icon). The main content area is titled "Servers connected to the cluster" and lists four entries:

Server ID	Database	Status	Details
2d8488bd21da_tvx	default	0 primaries, 0 secondaries	2d8488bd21da, up for 12 minutes
ed6ab6a3ed52_8hm	default	0 primaries, 0 secondaries	ed6ab6a3ed52, up for 7 minutes
b865c0a6e719_1bq	default	0 primaries, 0 secondaries	b865c0a6e719, up for 7 minutes
8dd016365e57_i05	default	0 primaries, 0 secondaries	8dd016365e57, up for 17 minutes

At the bottom of the page, there are links to Documentation, API, Google Groups, #rethinkdb on freenode, Github, and Community.

Use this interface and create a database called `imagin` in the **Tables** section.

## Running our service

The only part missing is our main service. We need to rebuild it again. If you change the source code to use RethinkDB again, you have to change the package.json file in order to have different dependencies. We need to rebuild this using our Dockerfile. To avoid introducing another theme, which is Docker Volumes, let's change our RethinkDB connection to not depend on the settings file:

```
rethinkdb.connect({ host: "database", db: "imagini" }, (err, db) => {
```

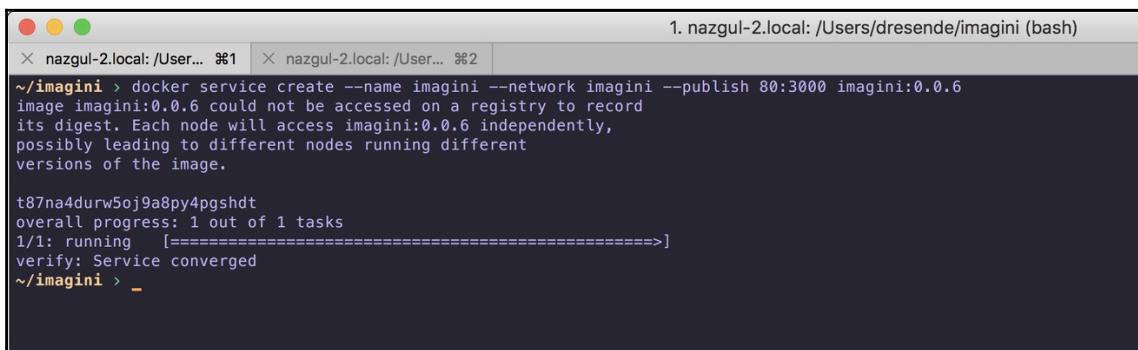
This way, we can avoid mounting a settings file altogether. Now, let's just increment the version and build it on both tabs:

```
docker build -t imagini:0.0.6 .
```

Now, the only thing missing is creating our service:

```
docker service create --name imagini --network imagini --publish 80:3000
imagini:0.0.6
```

In just a few seconds, we have our service running:

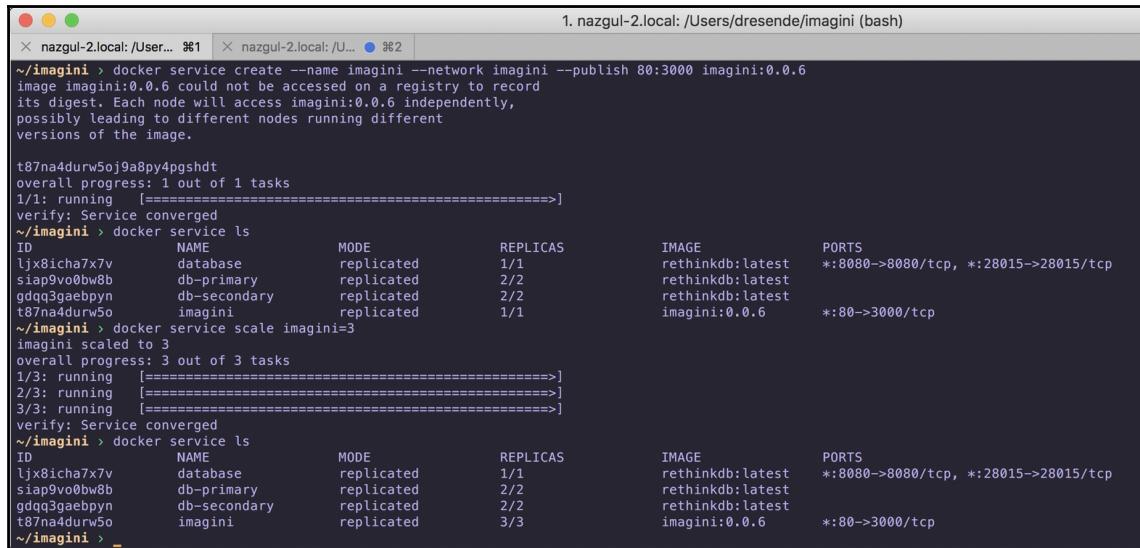


A screenshot of a macOS terminal window titled 'nazgul-2.local: /Users/dresende/imagini (bash)'. It shows the command 'docker service create --name imagini --network imagini --publish 80:3000 imagini:0.0.6' being run. The terminal output indicates that the image 'imagini:0.0.6' could not be accessed on a registry to record its digest, so each node will access 'imagini:0.0.6' independently. It also shows the overall progress of the task, with 1 out of 1 tasks running, and a progress bar indicating convergence.

```
nazgul-2.local: /Users/dresende/imagini (bash)
~/imagini > docker service create --name imagini --network imagini --publish 80:3000 imagini:0.0.6
image imagini:0.0.6 could not be accessed on a registry to record
its digest. Each node will access imagini:0.0.6 independently,
possibly leading to different nodes running different
versions of the image.

t87na4durw5oj9a8py4pgshdt
overall progress: 1 out of 1 tasks
1/1: running [=====>]
verify: Service converged
~/imagini > _
```

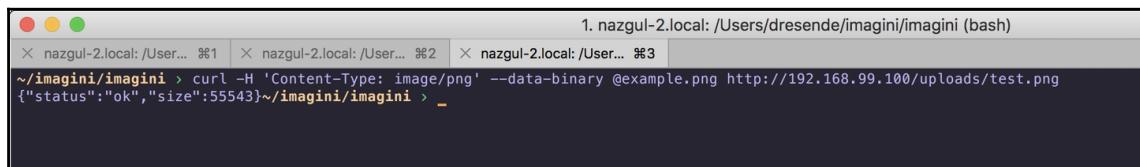
We can even scale it right away and have three instances:



```
1. nazgul-2.local: /Users/dresende/imagini (bash)
~ nazgul-2.local: /User... %1  x nazgul-2.local: /User... %2  x nazgul-2.local: /User... %3
~/imagini > docker service create --name imagini --network imagini --publish 80:3000 imagini:0.0.6
image imagini:0.0.6 could not be accessed on a registry to record
its digest. Each node will access imagini:0.0.6 independently,
possibly leading to different nodes running different
versions of the image.

t87na4durw5oj9a8py4pgshdt
overall progress: 1 out of 1 tasks
1/1: running [=====]
verify: Service converged
~/imagini > docker service ls
ID          NAME      MODE      REPLICAS      IMAGE      PORTS
ljq8icha7x7v    database  replicated   1/1        rethinkdb:latest
siap9vo0bw8b    db-primary replicated   2/2        rethinkdb:latest
gdqg3gaebpyn   db-secondary replicated  2/2        rethinkdb:latest
t87na4durw5o    imagini   replicated   1/1        imagini:0.0.6      *:80->3000/tcp
~/imagini > docker service scale imagini=3
imagini scaled to 3
overall progress: 3 out of 3 tasks
1/3: running [=====
2/3: running [=====
3/3: running [=====
verify: Service converged
~/imagini > docker service ls
ID          NAME      MODE      REPLICAS      IMAGE      PORTS
ljq8icha7x7v    database  replicated   1/1        rethinkdb:latest
siap9vo0bw8b    db-primary replicated   2/2        rethinkdb:latest
gdqg3gaebpyn   db-secondary replicated  2/2        rethinkdb:latest
t87na4durw5o    imagini   replicated   3/3        imagini:0.0.6      *:80->3000/tcp
~/imagini > _
```

We can see that it's exposed on port 80. As we did previously, we can see that the service is up and running. I'll just upload the example image again as this is a clean service and it has no images yet:



```
1. nazgul-2.local: /Users/dresende/imagini/imagini (bash)
~ nazgul-2.local: /User... %1  x nazgul-2.local: /User... %2  x nazgul-2.local: /User... %3
~/imagini/imagini > curl -H 'Content-Type: image/png' --data-binary @example.png http://192.168.99.100/uploads/test.png
>{"status":"ok","size":55543}~/imagini/imagini > _
```

You can see it on the RethinkDB administration interface. And even if we use the other swarm address, we see that the image is now available to download:

