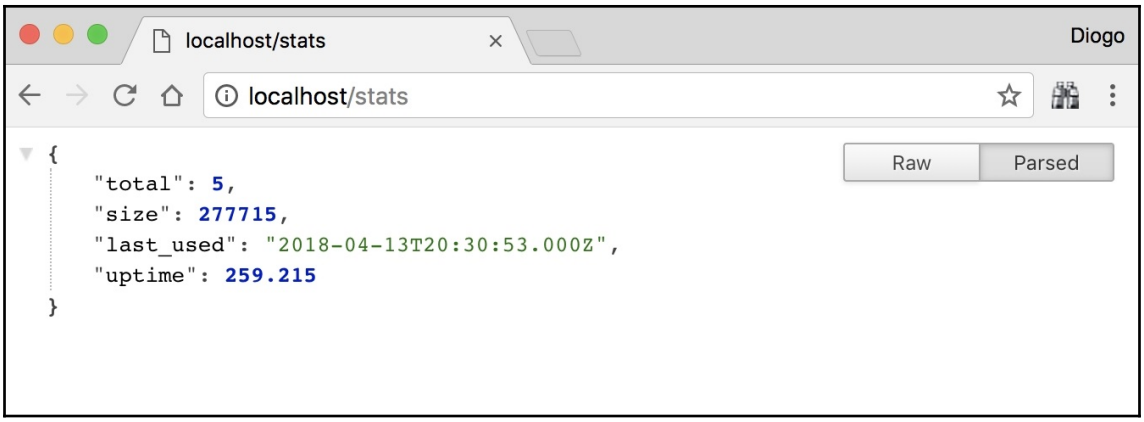


We can now manage our container in a similar fashion to the service management on the current operative systems. You can start, stop, and restart it:

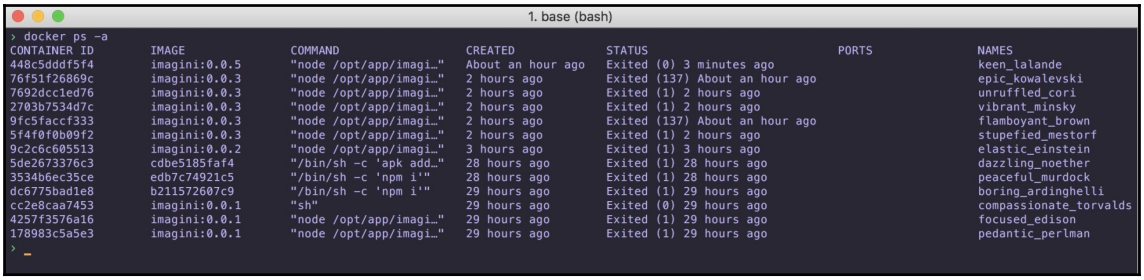
Every time you change one or more container's states, Docker returns the container IDs that have changed state. This allows you to, for example, to script actions and pipe the output of one command to the next.

Cleaning containers

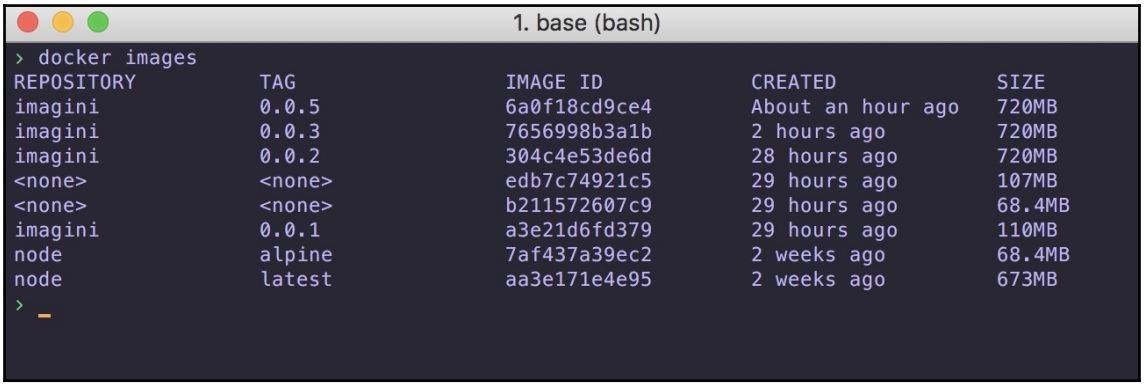
Containers, when stopped, are not removed by default. Let's stop our container and list the currently running containers:

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:



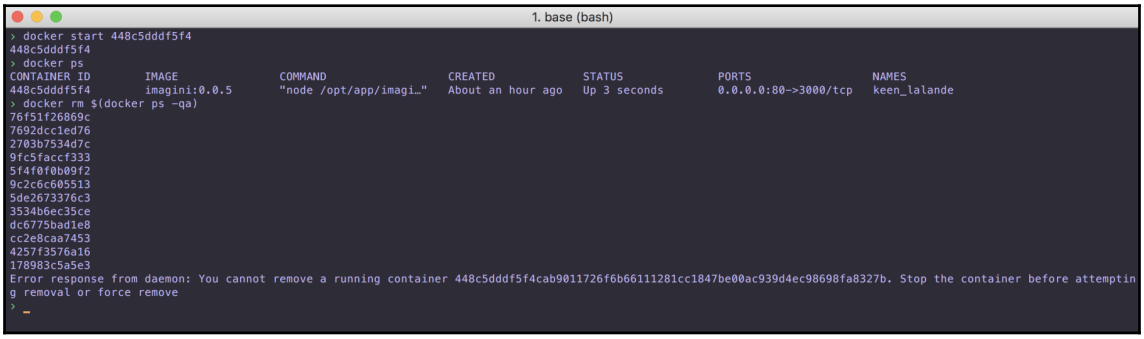
Wow, that's a lot of containers. We can see that we have containers for practically all the images we built, and that means the images are also available and using our disk space:



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.

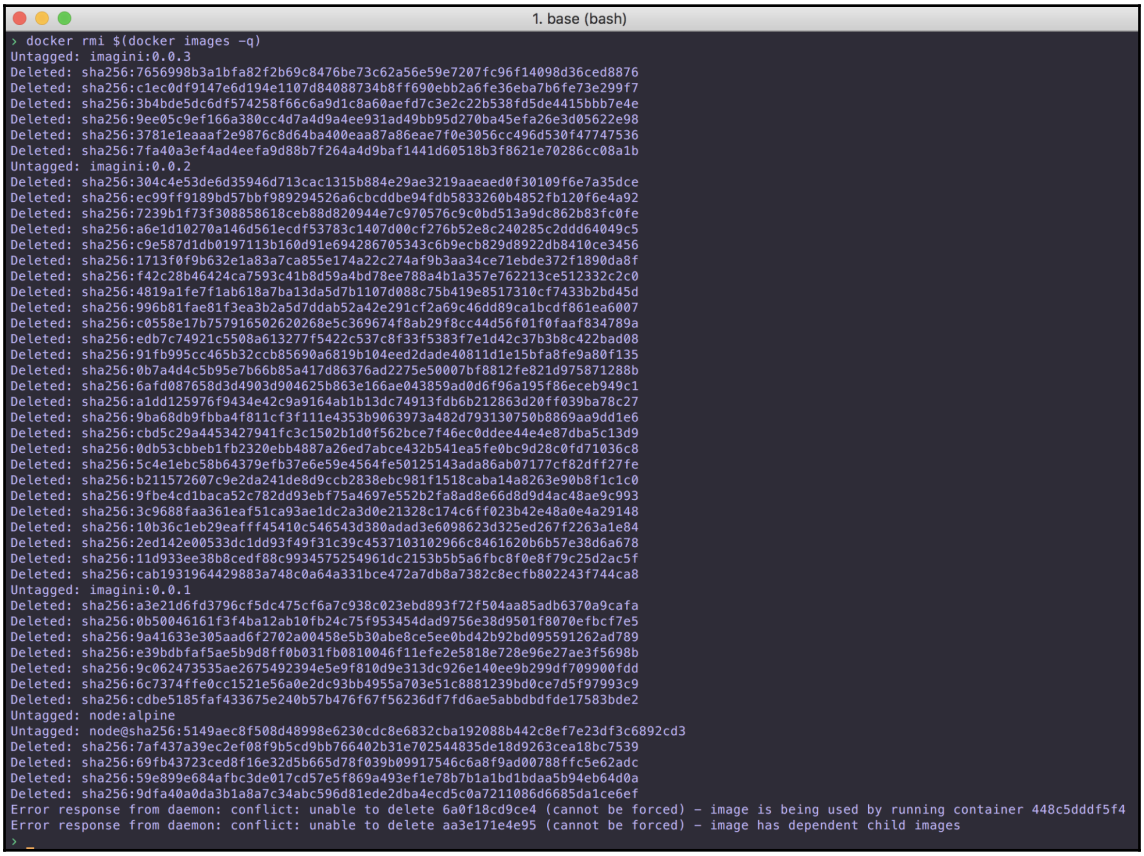
But before removing the images, we need to remove the containers, or Docker will deny removing them as they're being used. As a dirty workaround, we can use a useful parameter from `docker ps`, which returns only the container IDs and passes those to the remove command. Docker won't remove any running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

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



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:



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:

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:

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`:

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

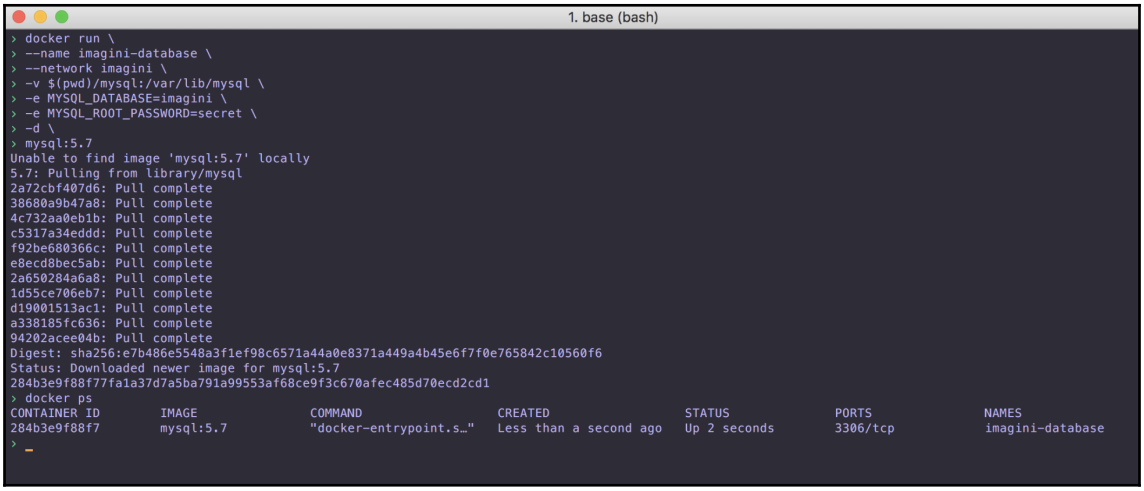
There are now four networks, one of which is the one we created. The other three are:

- **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:

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:

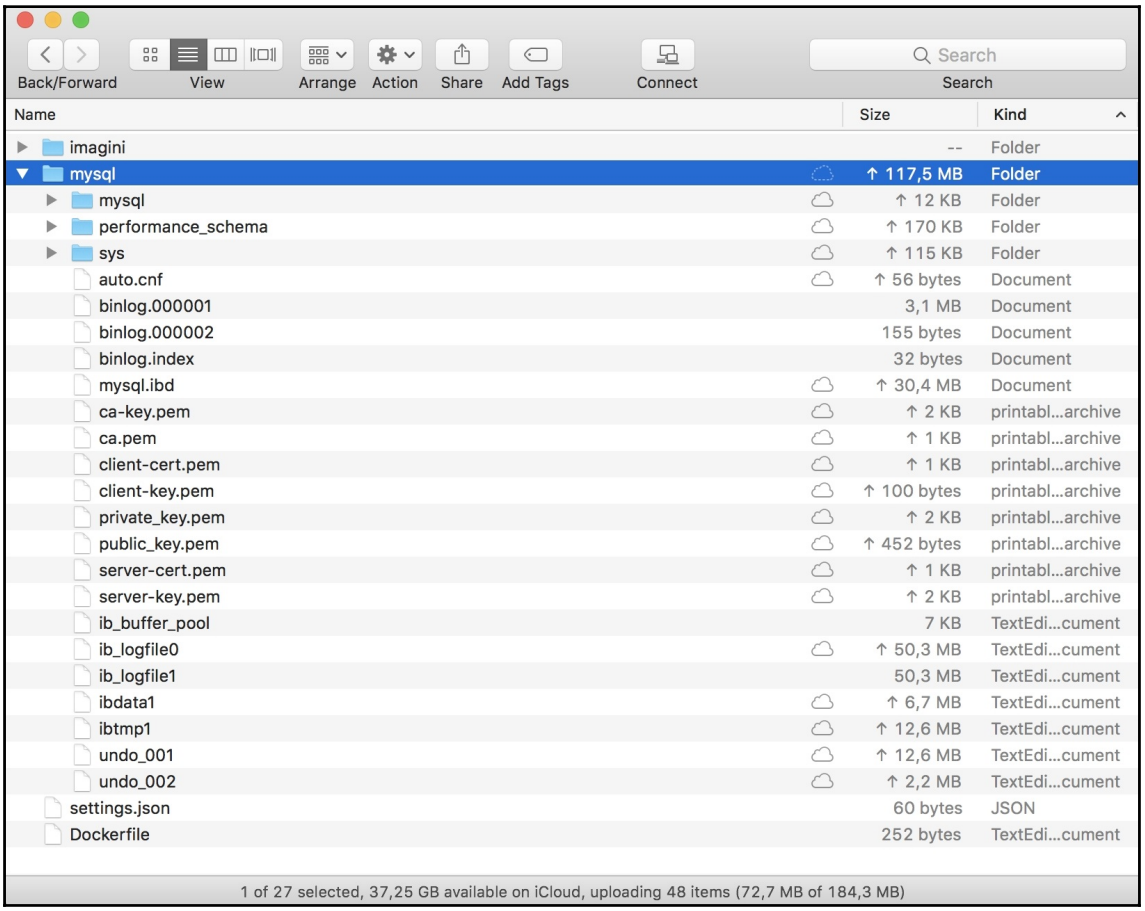


Docker automatically downloaded the latest MySQL version. We now have a database server called imagini-database running on our imagini network. Notice we didn't assign any of its ports to the host because we actually don't need to access it outside our custom network.

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.

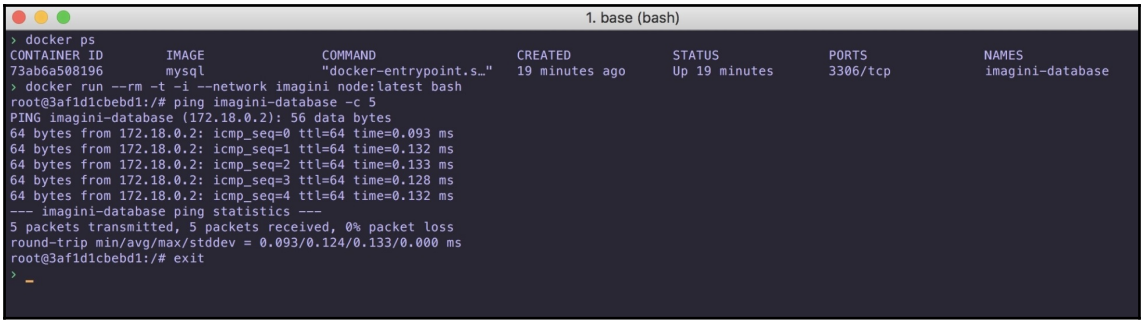
This is specific to this image and you can read more about it on the official Docker Hub page. We can confirm that the server is up and running using our local folder by just looking at its content:



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:



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:

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:

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:

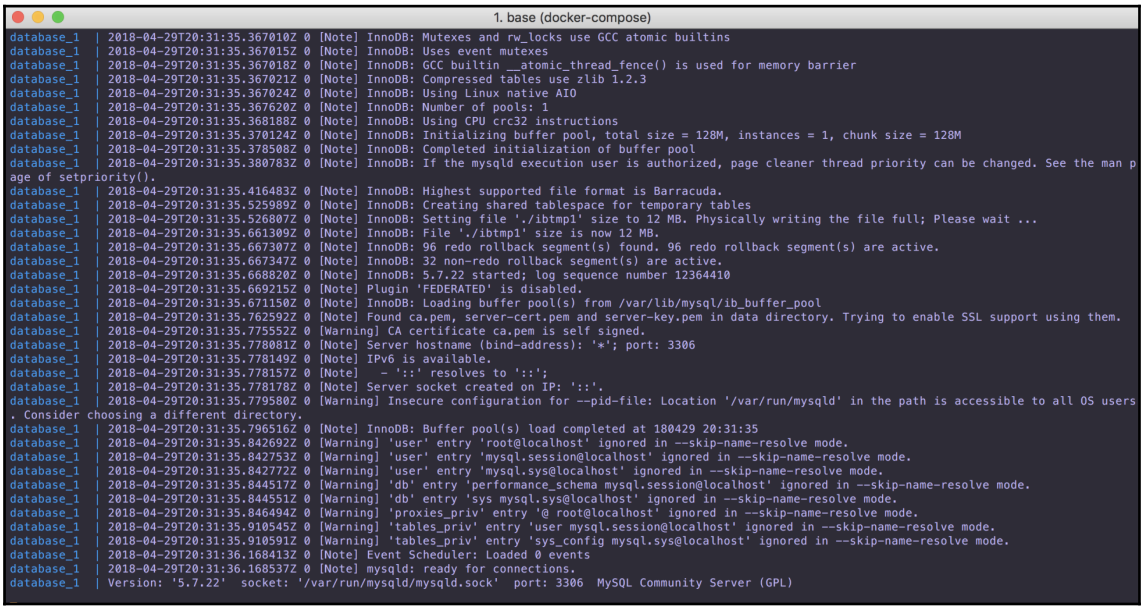
```
    service:
        image: imagini:0.0.5
        networks:
        - imagini
        volumes:
        - ${PWD}/settings.json:/opt/app/settings.json
        ports:
        - "80:3000"
        restart: on-failure
```

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.

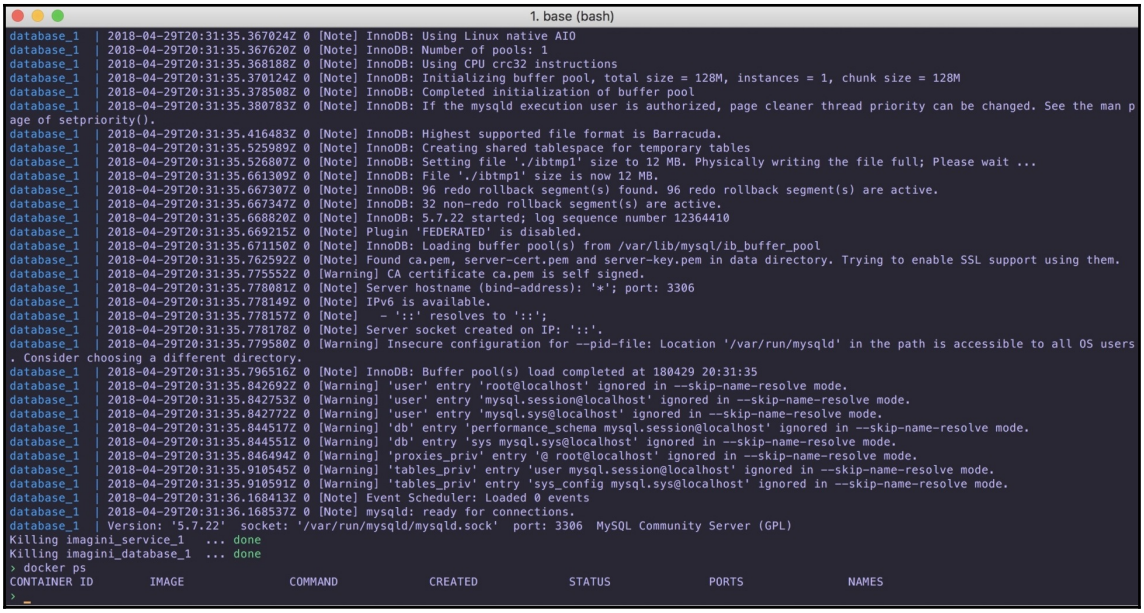
As we're composing a Docker Compose project, we can avoid using `imagini` on the service names, and so we need to once again change our `settings` file to something like this:

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

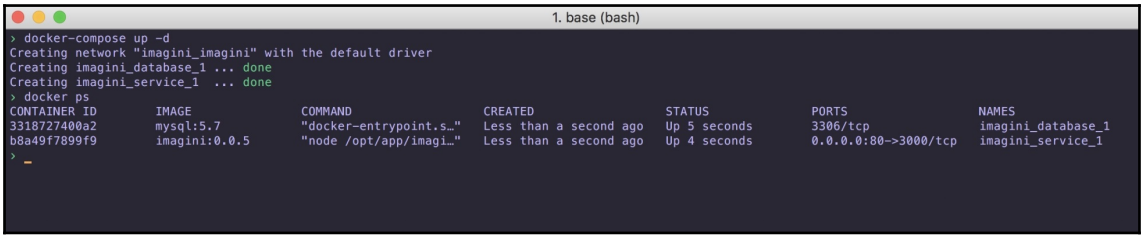
We can now try it out by running `docker-compose up`. This will deploy our containers under the `imagini` project, which is the folder name:



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:



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

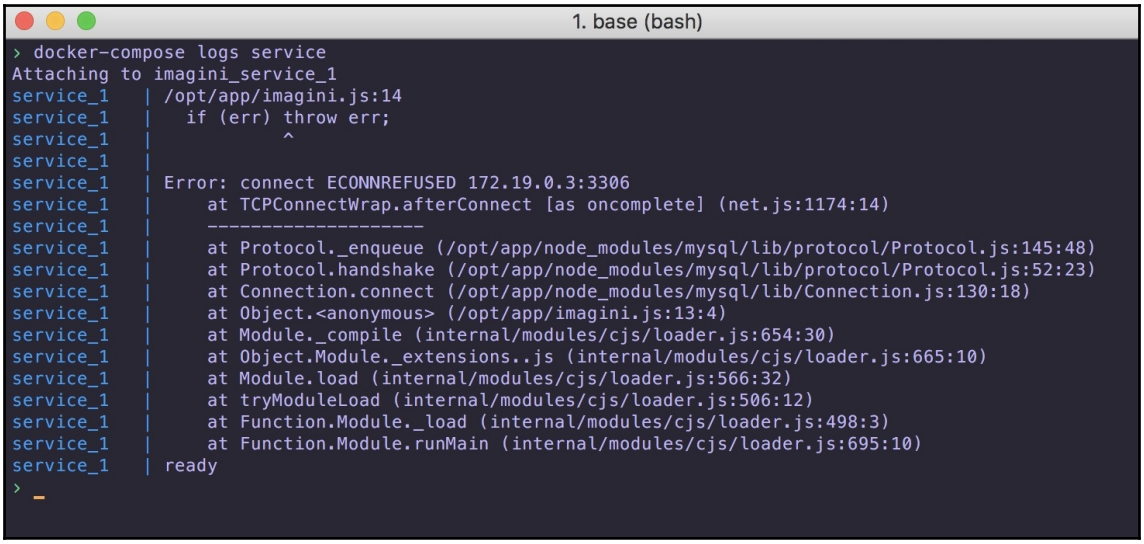


Great! We can start our service with everything it needs in a single command. You could just share this folder with a co-worker and he or she could have the service up and running in a matter of seconds.

Mastering Docker Compose

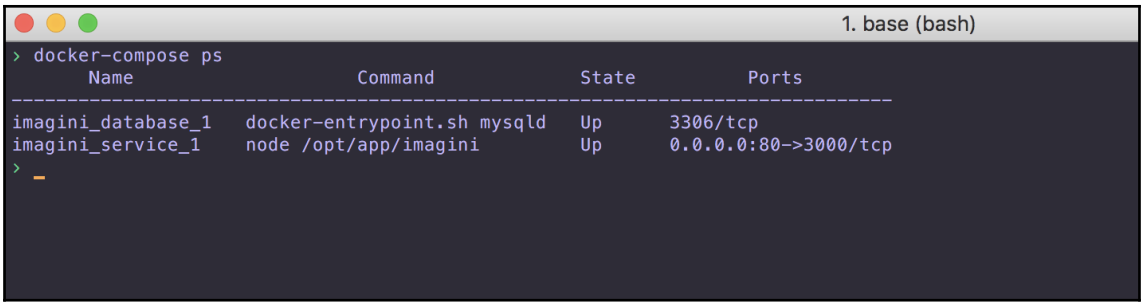
Now that we have our services up and running, we need some more information to manage these services together, and more specifically, to control and monitor its state.

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:



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:



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:

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.

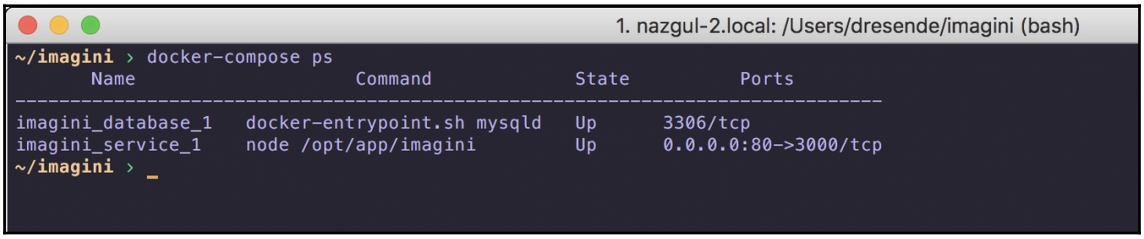
Services are usually reachable by a DNS address. You need to be able to control address name resolution, preferably by geography, to point your customers to the best instances. You can point directly to instances or have proxies monitor a couple of nearby instances and proxy the traffic based on policies.

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 busier 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:



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.

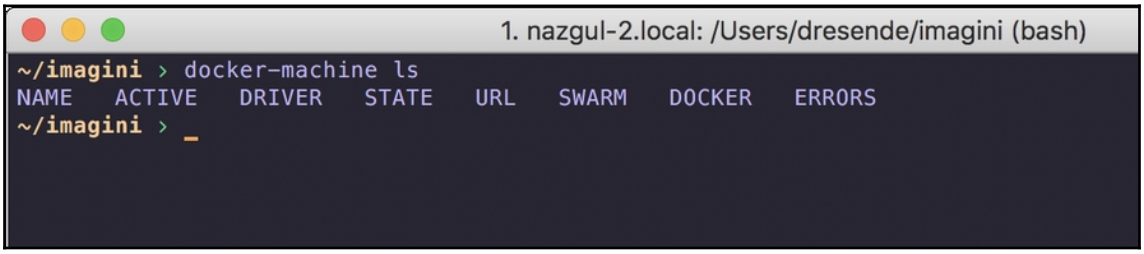
Docker swarm is another component of the Docker engine that enables you to create a cluster of hosts to run your microservice instances while orchestrating all of this using a simple tool from any of the hosts in your cluster.

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
```

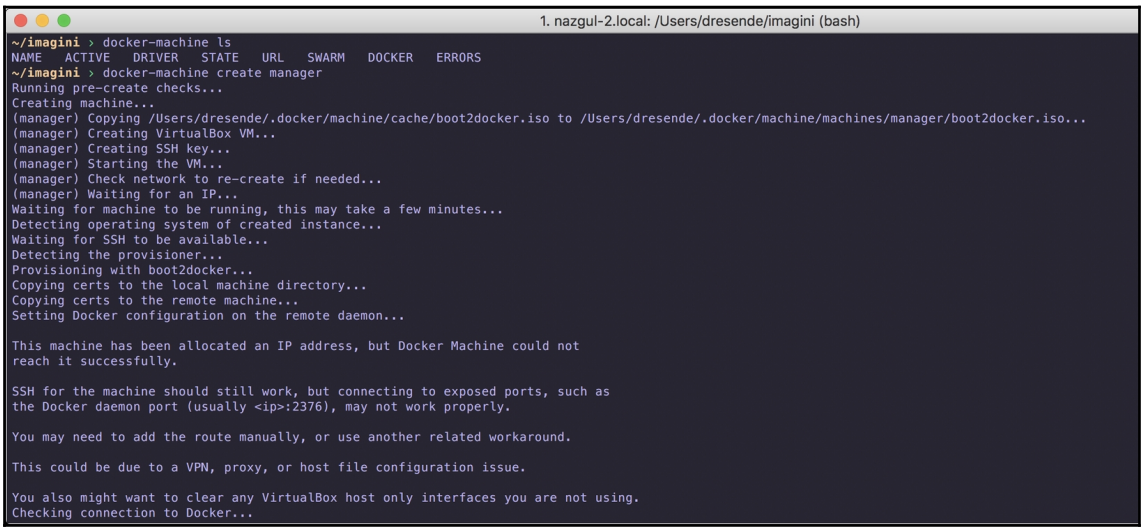
You should see an output as follows:



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

```
docker-machine create manager
```

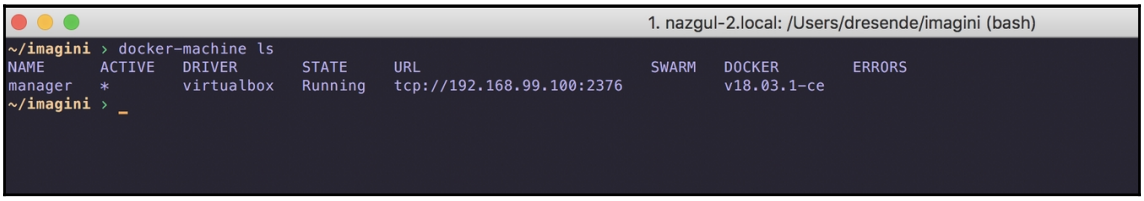
Docker will create the new machine, assign a new IP address, and prepare the Docker engine inside the machine:



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:

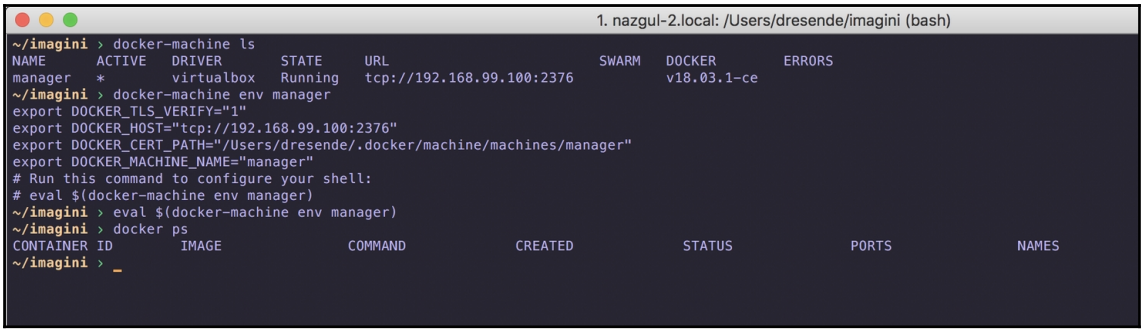


The Docker engine has an exposed API that enables you to manage a Docker host using the Docker commands from another host. The URL column indicates the API address to manage that specific host.

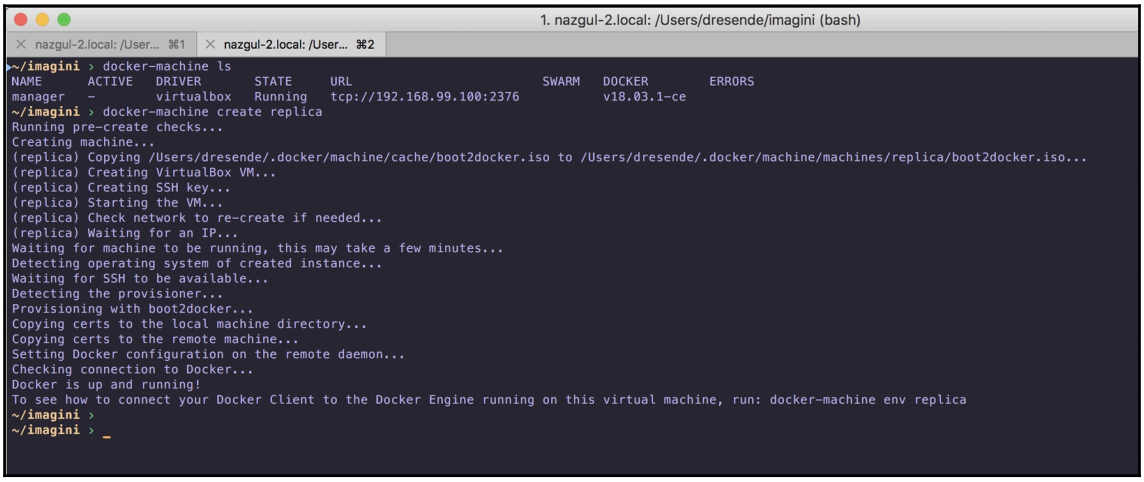
We can use a little helper command to configure our Docker command to manage this new host. Just type the following command:

```
docker-machine env manager
```

And follow the instructions:



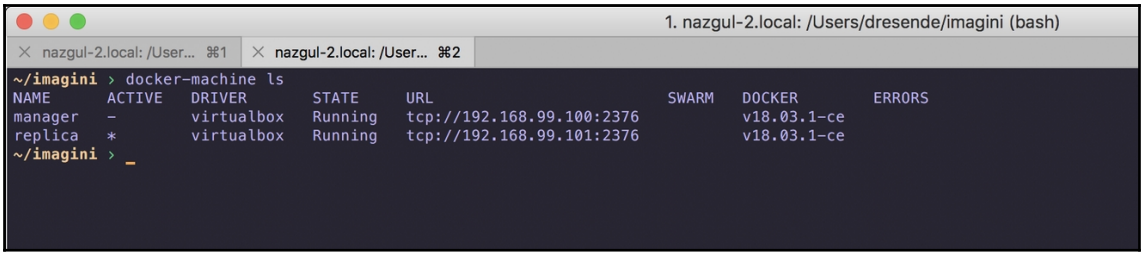
Now, let's keeps this Terminal untouched and open a new tab so that we can create a second machine. Let's call it replica, which will hold a second instance of our microservice:



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

```
docker-machine ls
```

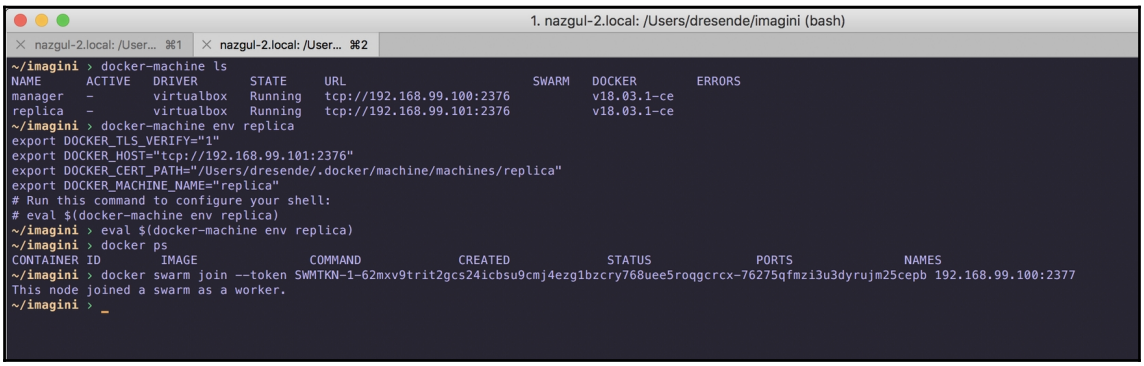
Both of our machines should be ready and running:



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:

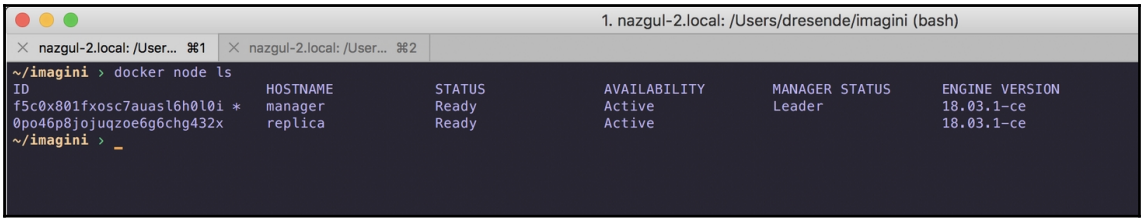
Docker created our swarm and gave us instructions on how to make other hosts join the swarm using a kind of secret token. If you want to add more nodes later on, save that command.

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



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:



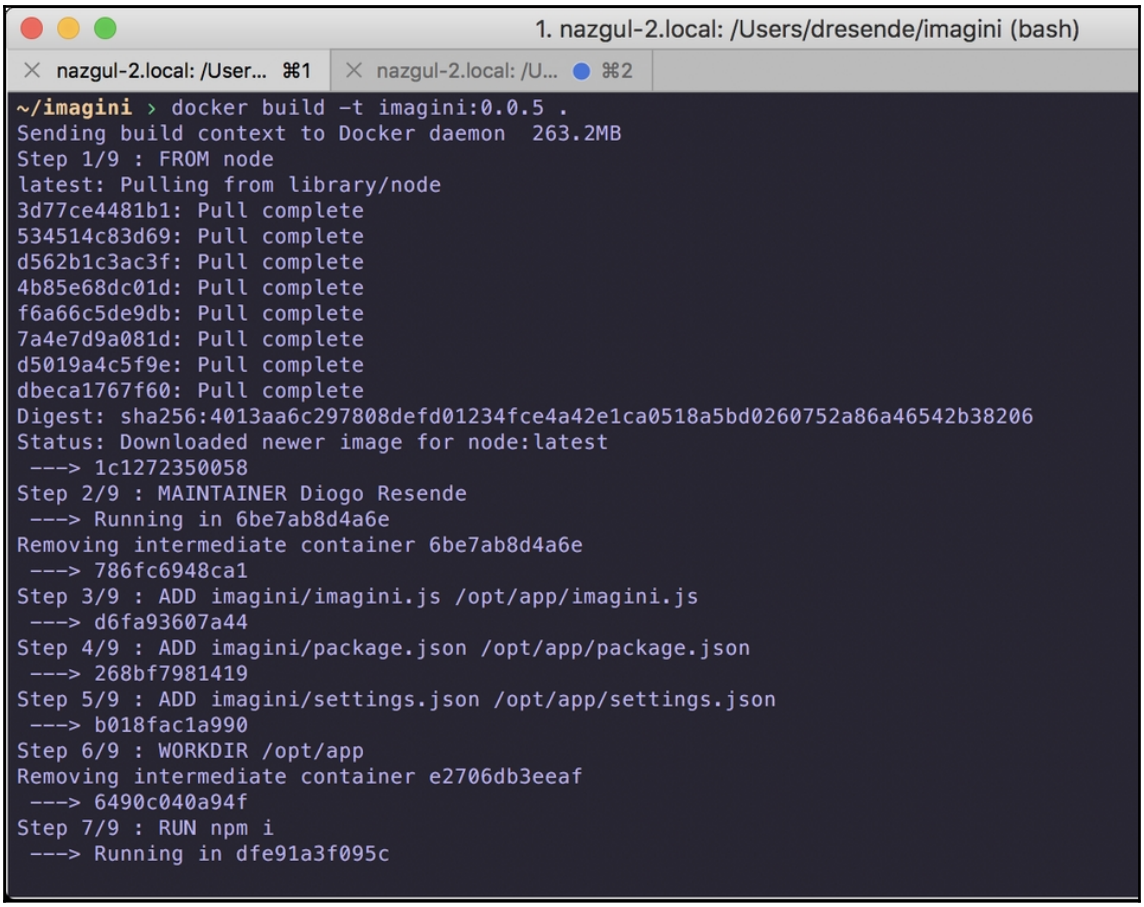
Deploying to swarm

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:



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:

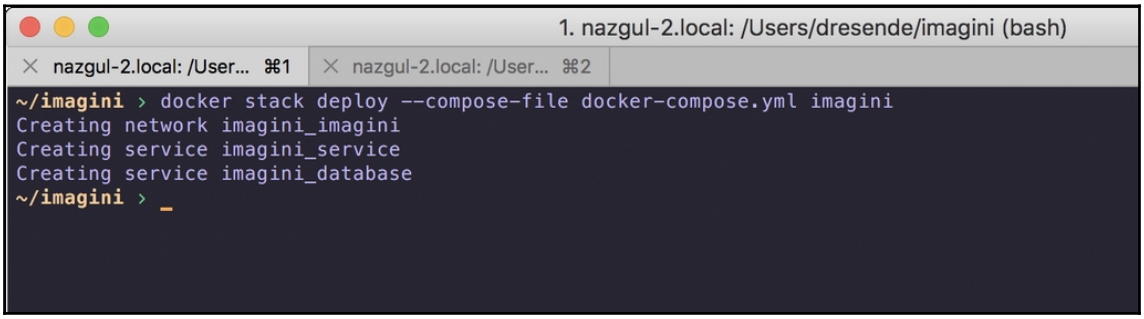
```
version: "3"
networks:
    imagini:
services:
    database:
        image: mysql:5.7
        networks:
        - imagini
        volumes:
        - /var/lib/mysql:/var/lib/mysql
        ports:
        - "3306:3306"
        environment:
            MYSQL_DATABASE: imagini
            MYSQL_ROOT_PASSWORD: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

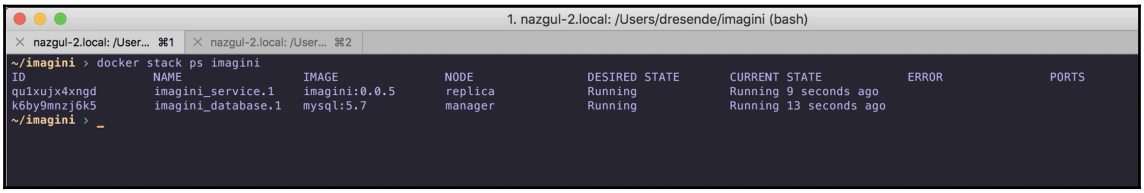
This should start the deployment. It will create the network on the swarm and then deploy the two services:



Wait a little bit and then run the following command:

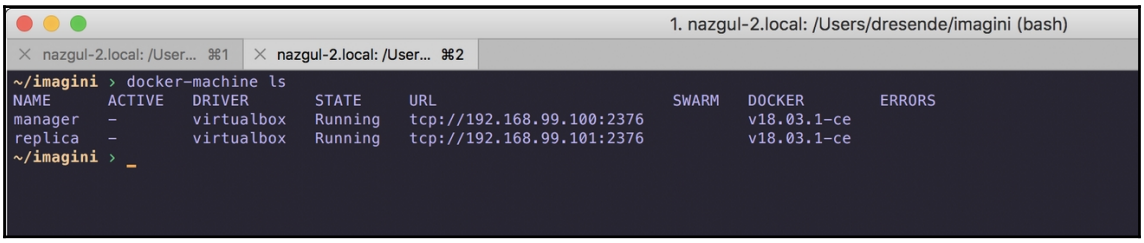
```
docker stack ps imagini
```

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

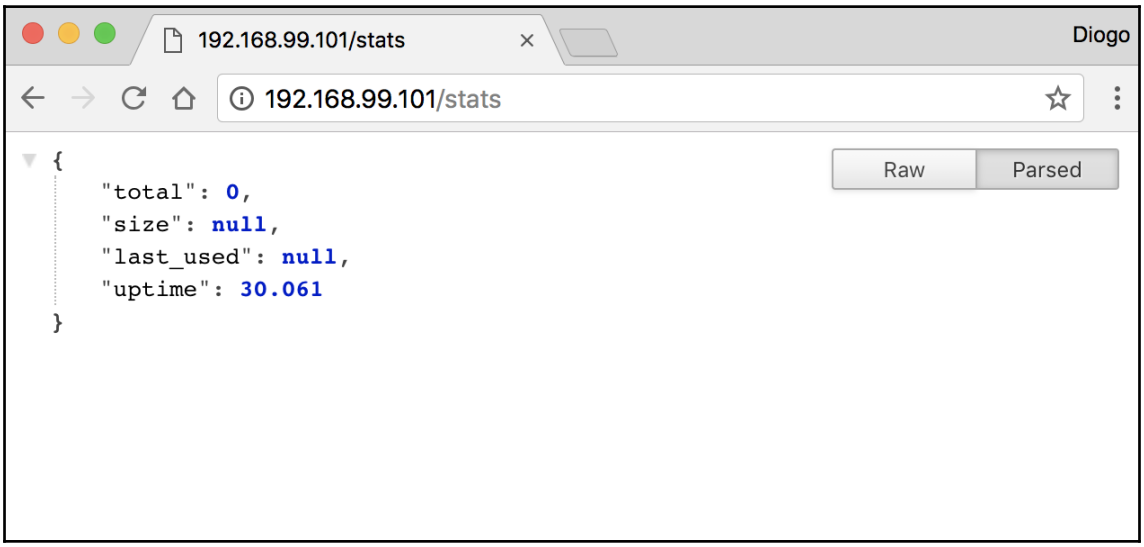


The only difference you may notice is that the name of the containers have a `.1` on the end of them and that a new `NODE` column indicates where it's running inside our swarm.

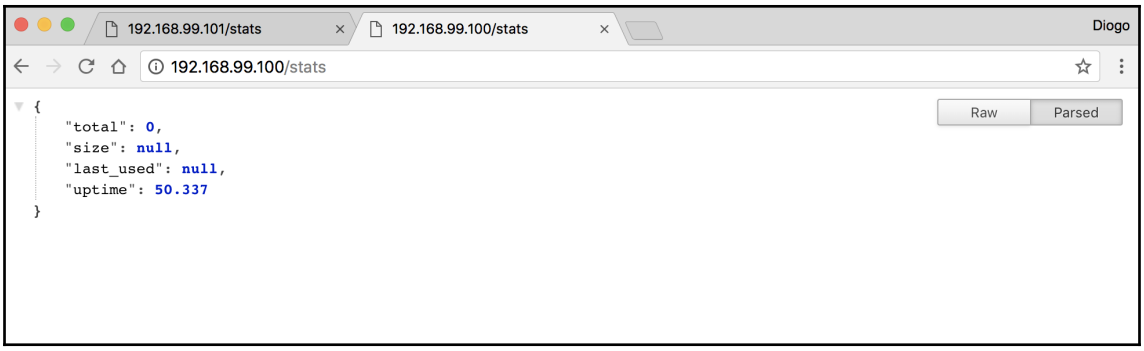
Because our containers are running in a swarm inside two virtual machines, we need to use the IP address of the node that has the service running on it. Looking at the previous screenshot, we can see that it's in the replica machine:



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:

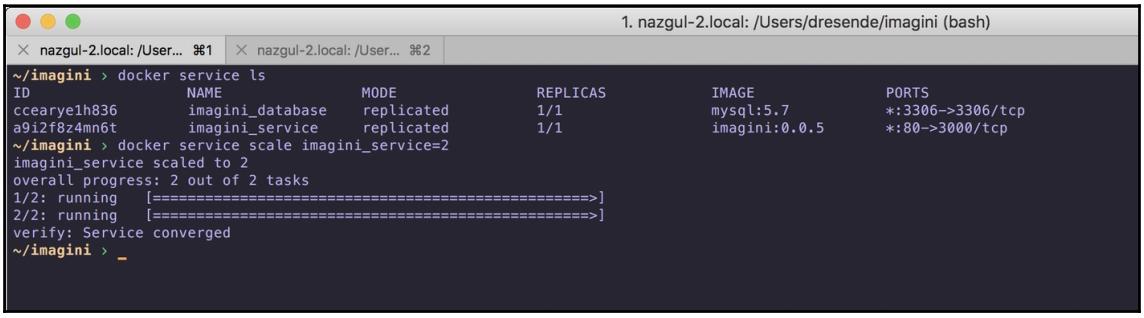


Now, this is something awesome. Notice that we only had one instance of each of our two services, and our `imagini` service was on the replica node. Although it's available from any of the swarm node addresses, if, for some reason, it fails, you wouldn't be able to reach it.

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
```

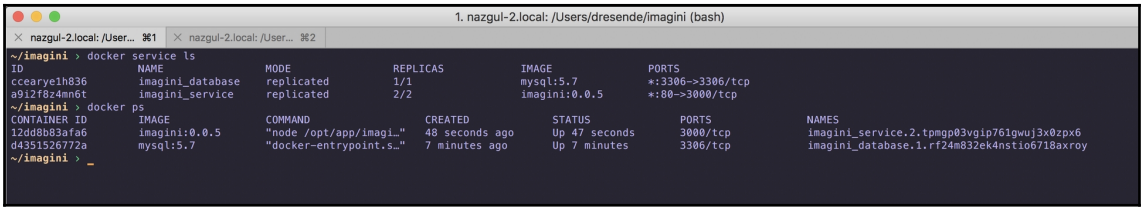
Docker will handle the deployment and check if everything goes as planned:



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

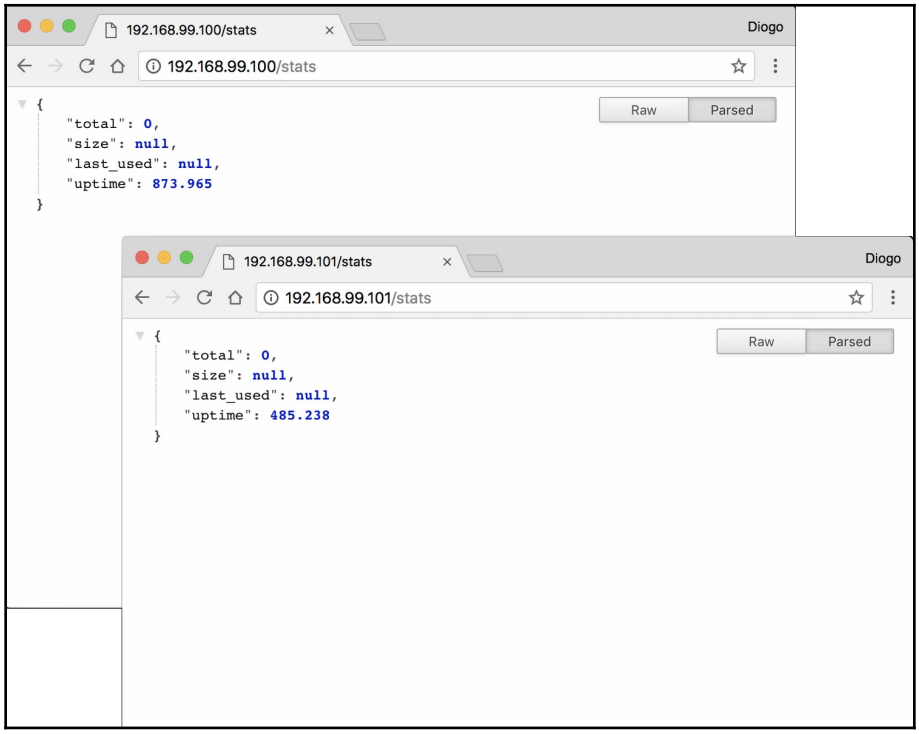
```
docker service ls
```

Notice how our service `2/2` replicas are running:



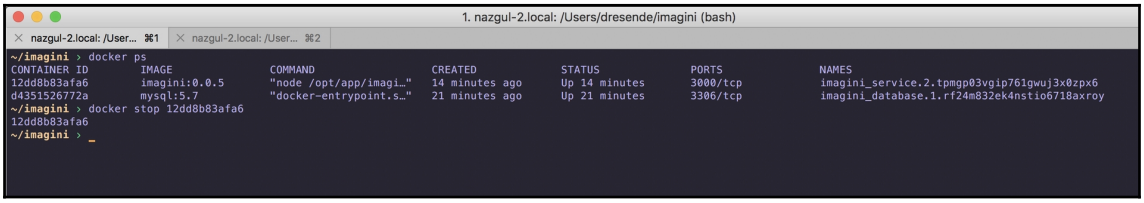
If you head to the browser, both addresses still work as expected, but in the background, we have two services running. You can notice this by looking at the `uptime` property.

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:

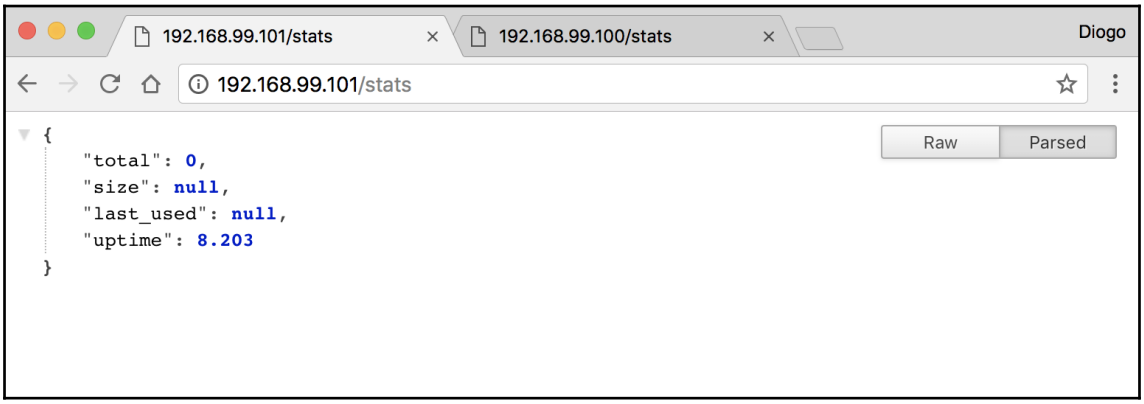


Notice how the swarm is not constant and does not give you the same instance for the same address. It keeps rotating it.

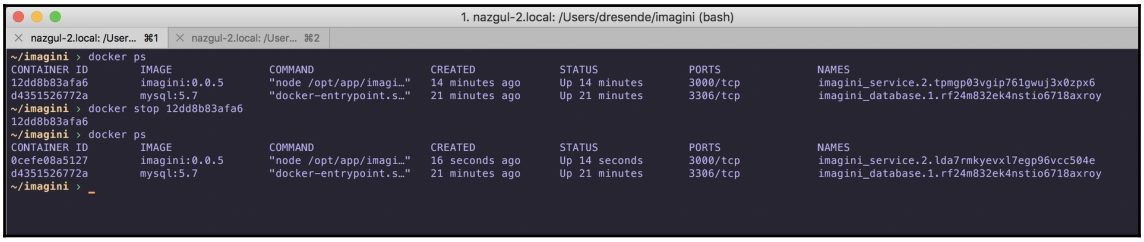
The swarm also monitors our container instances. For example, let's imagine you're working on a host and, by accident, stop a container:



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:



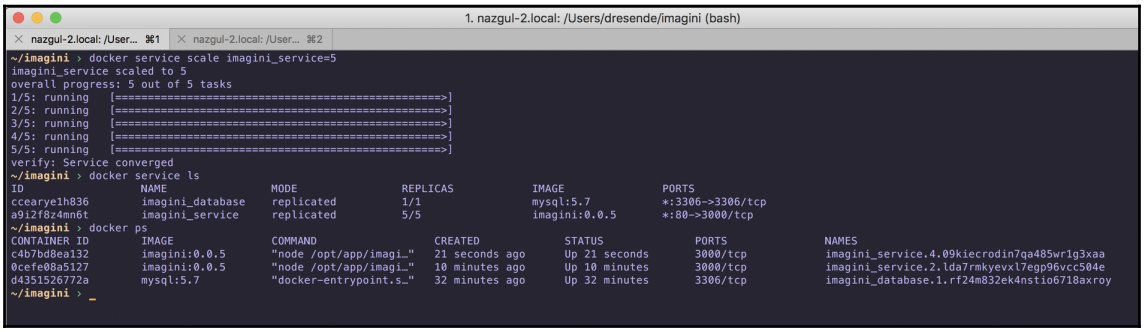
What happened to our `uptime`? It just went down to a few seconds. This is because the swarm just noticed our container stopping and restarted it. If you look at the Docker containers again, it's still running. Well, actually it restarted:



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:



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
```

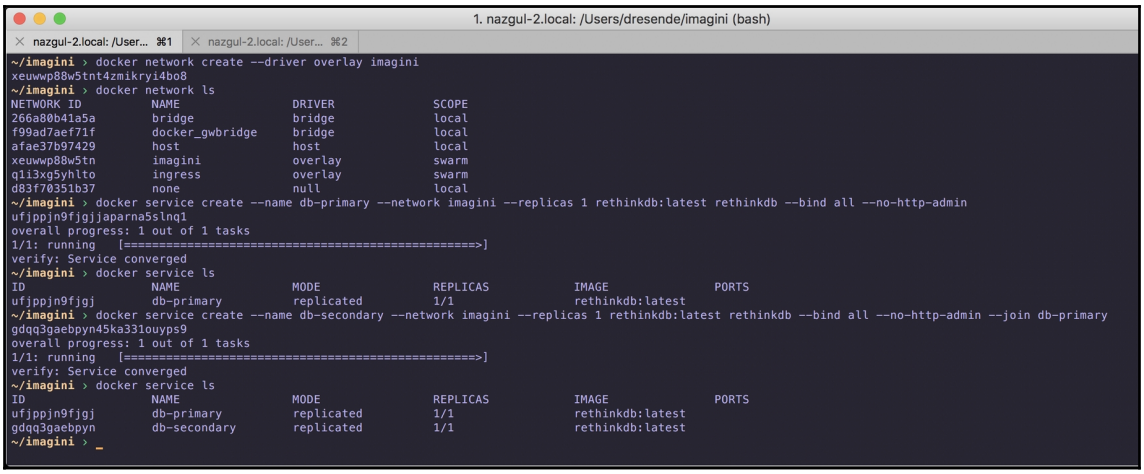
Then, create `db-primary` with only `1` replica as we're removing it later on:

```
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:



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

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

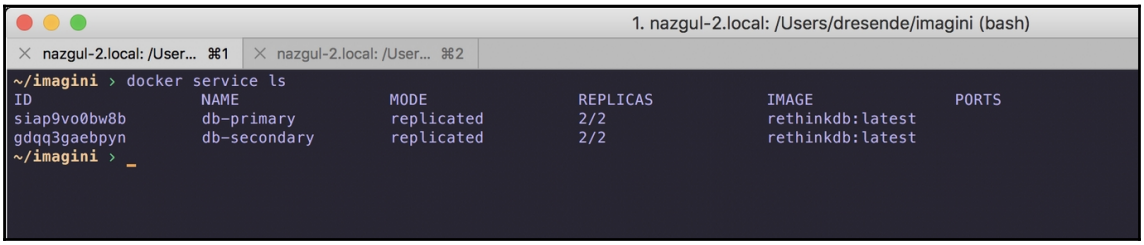
Then, remove the `db-primary` and recreate it:

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

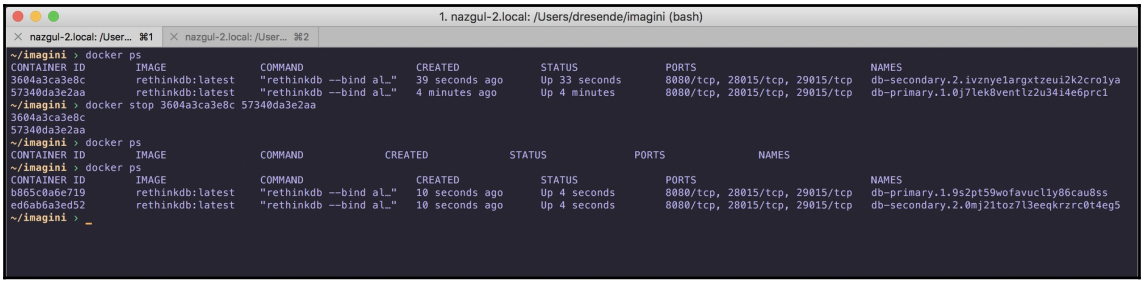
Then, scale it to two instances:

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

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



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:



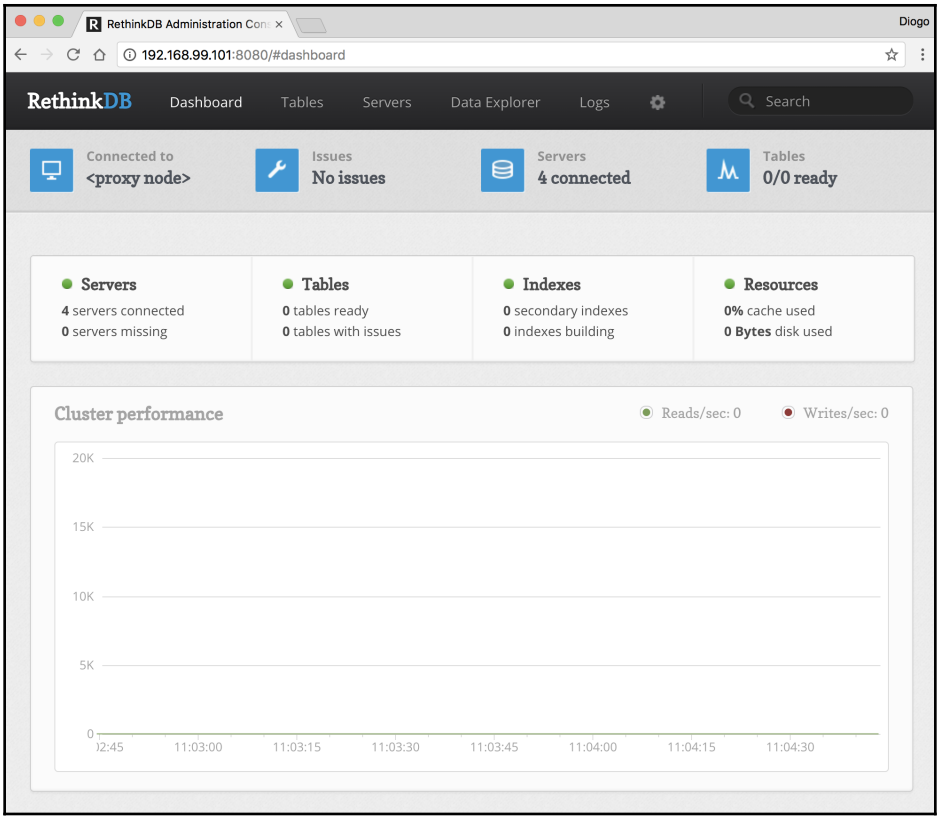
Now that we have our database cluster ready, we just need a proxy to be able to access the cluster:

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

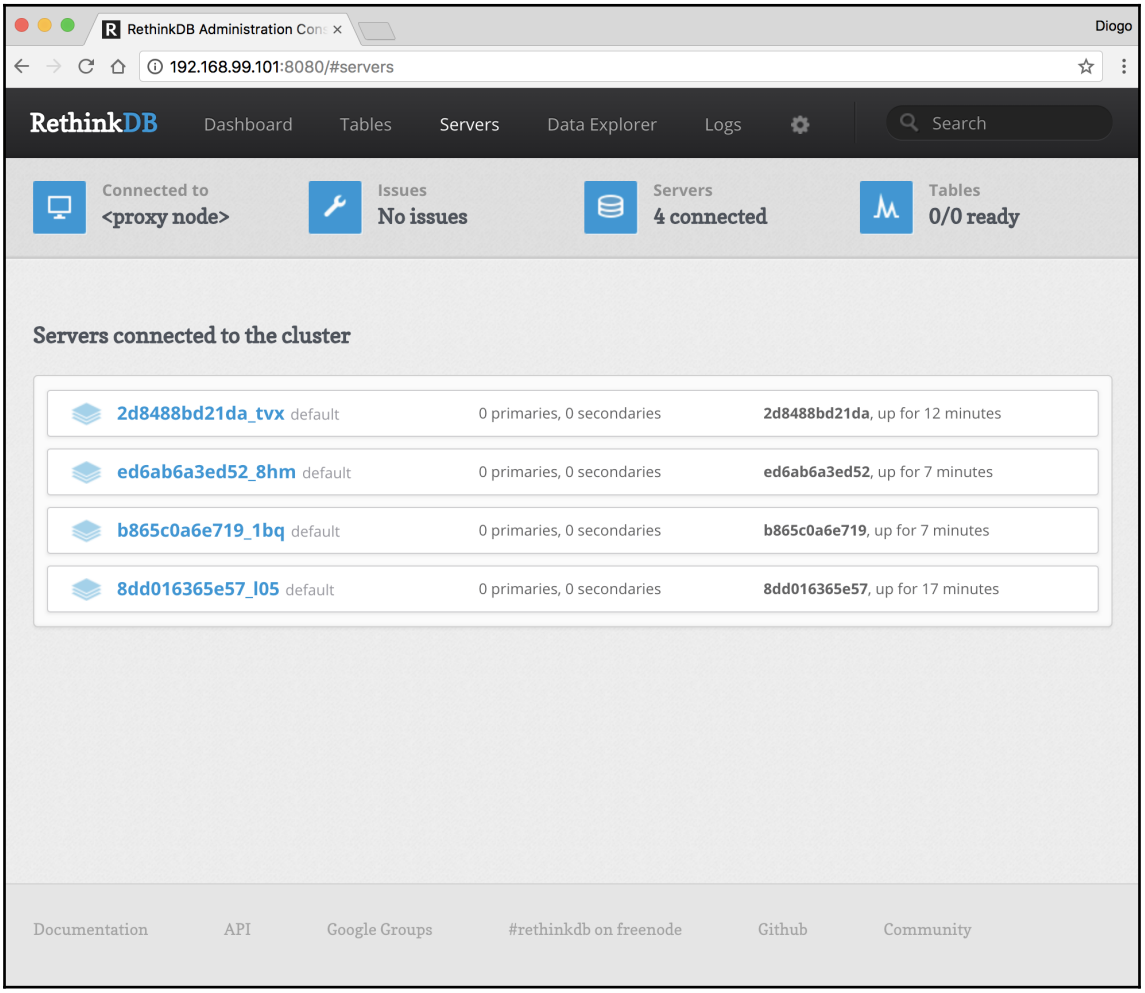
If we look at the services, we can now see that we have our four cluster nodes and a proxy called database, which we'll use to manage and access the data:

```
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
ljx8icha7x7v9owf5vibma2h9
overall progress: 1 out of 1 tasks
1/1: running   [==================================================>]
verify: Service converged
~/imagini › docker service ls
ID            NAME           MODE         REPLICAS   IMAGE              PORTS
ljx8icha7x7v  database       replicated   1/1        rethinkdb:latest   *:8080->8080/tcp, *:28015->28015/tcp
siap9vo0bw8b  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:



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:



Use this interface and create a database called `imagini` 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:

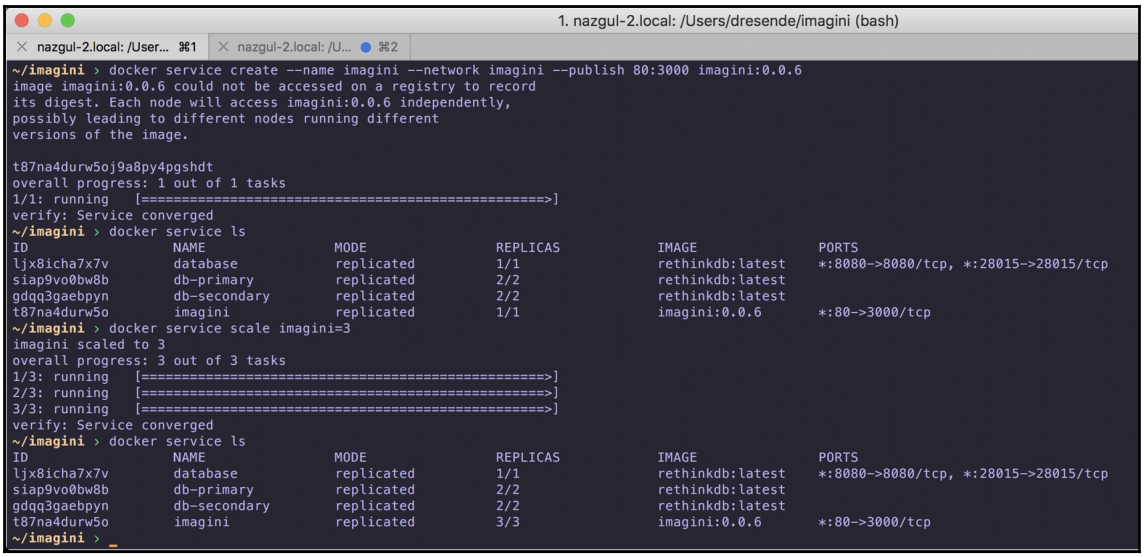
```
docker build -t imagini:0.0.6 .
```

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:

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



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:

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: