



The DZone Guide to Microservices: Breaking Down the Monolith



An API-First Approach for Microservices on Kubernetes

Akka: The Ideal Actor-Based Framework

Explore the Most Common Patterns for Microservices

Download My Free PDF

Site24x7


End to End **Cloud Monitoring** Platform

- ▲ Full Stack Visibility
- ▲ Designed to Scale
- ▲ Real Time Alerts



Cluster Using Docker in Docker

In the first part of this series, learn how to set up and deploy Docker in Docker, and experiment with reconfiguring the conditions of a single node.

by Magnus Larsson  MVB · Mar. 02, 18 · Cloud Zone

Create a continuous deployment pipeline with open source tools. Watch the screencast.

This blog series is about new features in Docker that I learned from visiting the DockerCon EU conference in Copenhagen, 16-19 October 2017, and in my daily work.

I will focus on news in the Docker Engine and surrounding tools, i.e. not go into details on the inner workings of container orchestrators, such as Docker in Swarm mode or Kubernetes. They are worth a blog series on their own.

However, I will use container orchestrators to demonstrate other features in Docker. For example, demonstrate how we can use Docker in Docker to quickly and easily setup a local container cluster on our development machines without blowing away the memory.

If you want to learn about how to develop microservices in Java or Go that can be deployed as Docker containers, take a look at the blog series, Building Microservices.

Background

Before deploying your Docker containers to production you most probably need a *Container Orchestrator*, such as Docker in Swarm mode or Kubernetes, to manage a cluster of nodes where your containers can be deployed (regardless of whether you are going to run them on premises or in the cloud). For a background see, for example, the blog post from containers to container orchestration.

Before deploying your Docker containers to a container orchestrator, you need to understand how your container orchestrator works. Even though you can setup a test instance in the cloud to get used to it, it is typically much more convenient and efficient to learn a complex software, such as a container orchestrator, if you initially can try it out locally.

Traditionally, setting up a local multi-node container orchestrator means spinning up a number of virtual machines locally and then setting up the container orchestrator on these nodes. It is known to both consume a lot of memory (a virtual machine with Docker running consumes at least 1 GB just to get started) and is also time consuming to setup.



Source

But, what if we can use a number of containers to act as our cluster nodes instead of fully fledged virtual machines?

We know for sure that they start much faster and have very little overhead in terms of memory!

The *only* tricky thing is that these containers needs to run Docker in themselves, i.e. run Docker in Docker.

Docker provides a. official Docker image for this specific purpose: Docker in Docker.

Words of warning: You should only use Docker in Docker for test purposes. See, for example, this blog post for details.

Let's try it out using Docker in Swarm mode!

We will use Docker for Mac acting both as the Swarm Manager and as the Docker Host for three Swarm Workers (i.e. that will run as containers).

Installation

ctop is like top , but for containers.

Setup

To be able to follow the startup sequence, launch the `ctop` tool in one terminal window:

```
1 # Monitor Master and Worker nodes
2 ctop
```

Next, in another terminal window, run commands to:

1. Ensure that Docker for Mac runs in Swarm mode, i.e. acts as a Swarm Master
2. Setup environment variables for the token to be used to join worker nodes and the IP address of the master node

Run the following commands:

```
1 # Init Swarm master
2 docker swarm init
3
```

Next, setup environment variables for what Docker version to use and how many worker nodes you want:

```
1  # Docker version
2  DOCKER_VERSION=17.09.1-ce-dind
3
4  # Number of workers
5  NUM_WORKERS=3
```

Now, we can launch the worker nodes as containers and ask them to join our Docker Swarm cluster:

```
1  # Run NUM_WORKERS workers with SWARM_TOKEN
2  for i in $(seq "${NUM_WORKERS}"); do
3  docker run -d --privileged --name worker-${i} --hostname=worker-${i} -p ${i}2375:2
4  docker --host=localhost:${i}2375 swarm join --token ${SWARM_TOKEN} ${SWARM_MASTER_
5  done
```

The first `docker run` command might take some time due to the required Docker image needing to be downloaded, otherwise it should only take a few seconds to setup the Swarm cluster!

Verify that you can see the worker nodes using `ctop` as ordinary containers in Docker for Mac:

**Note: The memory used per worker node is only some 30 MB!
To be compared to (at least) 1 GB required for an empty worker
node that runs as a virtual machine...**

To be able to monitor the state of the cluster you can start a Visualizer service as:

```
1  # Setup the visualizer
2  docker service create \
3    --detach=true \
4    --name=viz \
5    --publish=8000:8080/tcp \
6    --constraint=node.role==manager \
7    --mount=type=bind,src=/var/run/docker.sock,dst=/var/run/docker.sock \
8    dockersamples/visualizer
```

View

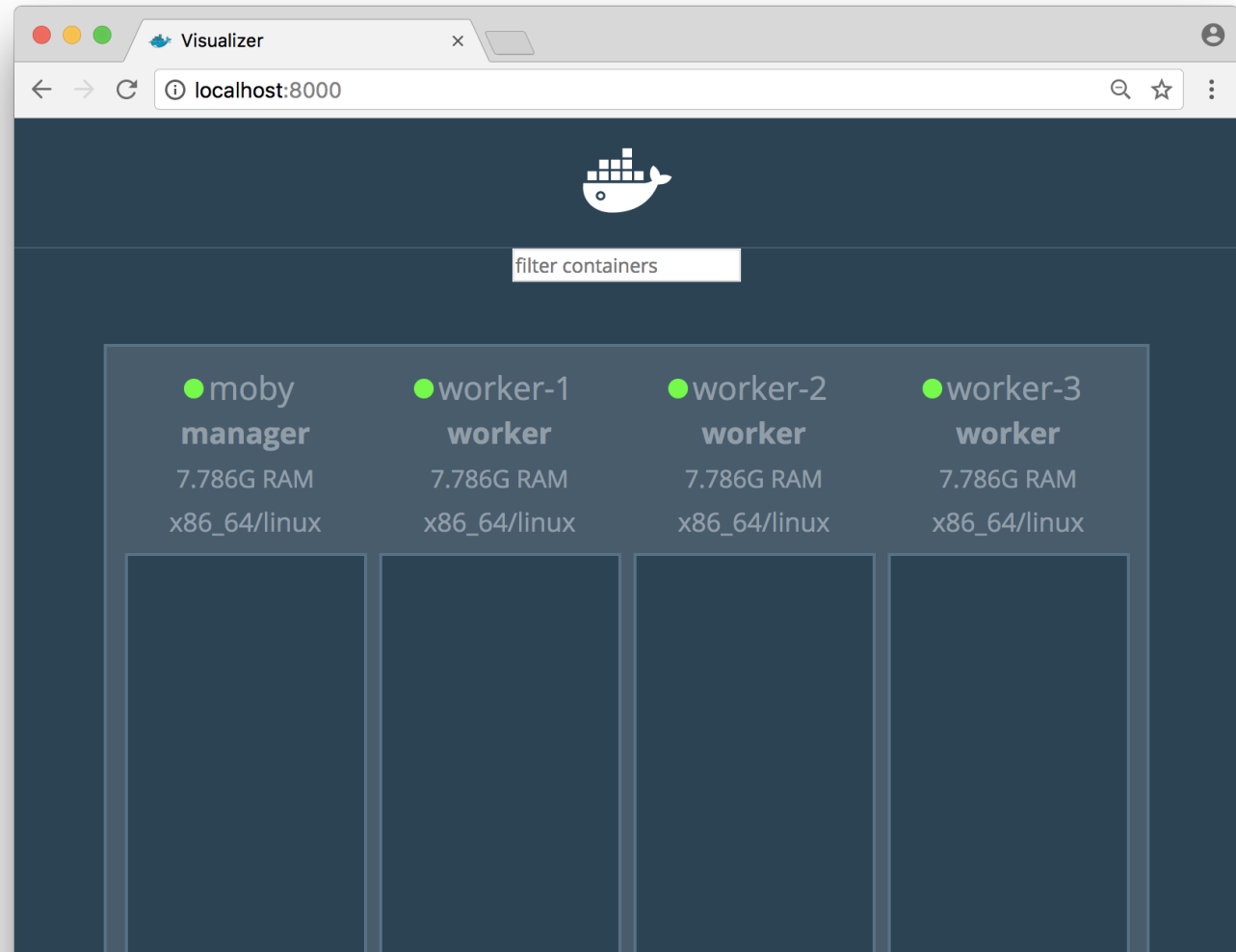
1	ID	HOSTNAME	STATUS	AVAILABILITY
2	p04lc6eikqz76rdzp9ehpejdx *	moby	Ready	Active
3	owpvm6mt8xpoxqk28mph81n0g	worker-1	Ready	Active
4	blf4o5lphy1k7ds7pn2sok1h9	worker-2	Ready	Active
5	wtia3kscr694577je4v5ryms4	worker-3	Ready	Active

Also, let's see if we have the Visualizer service up and running:

```
1 docker service ls
```

Expect output as follows:

1	ID	NAME	MODE	REPLICAS	IM
2	nv7uhth4k3vw	viz	replicated	1/1	do



Deploy

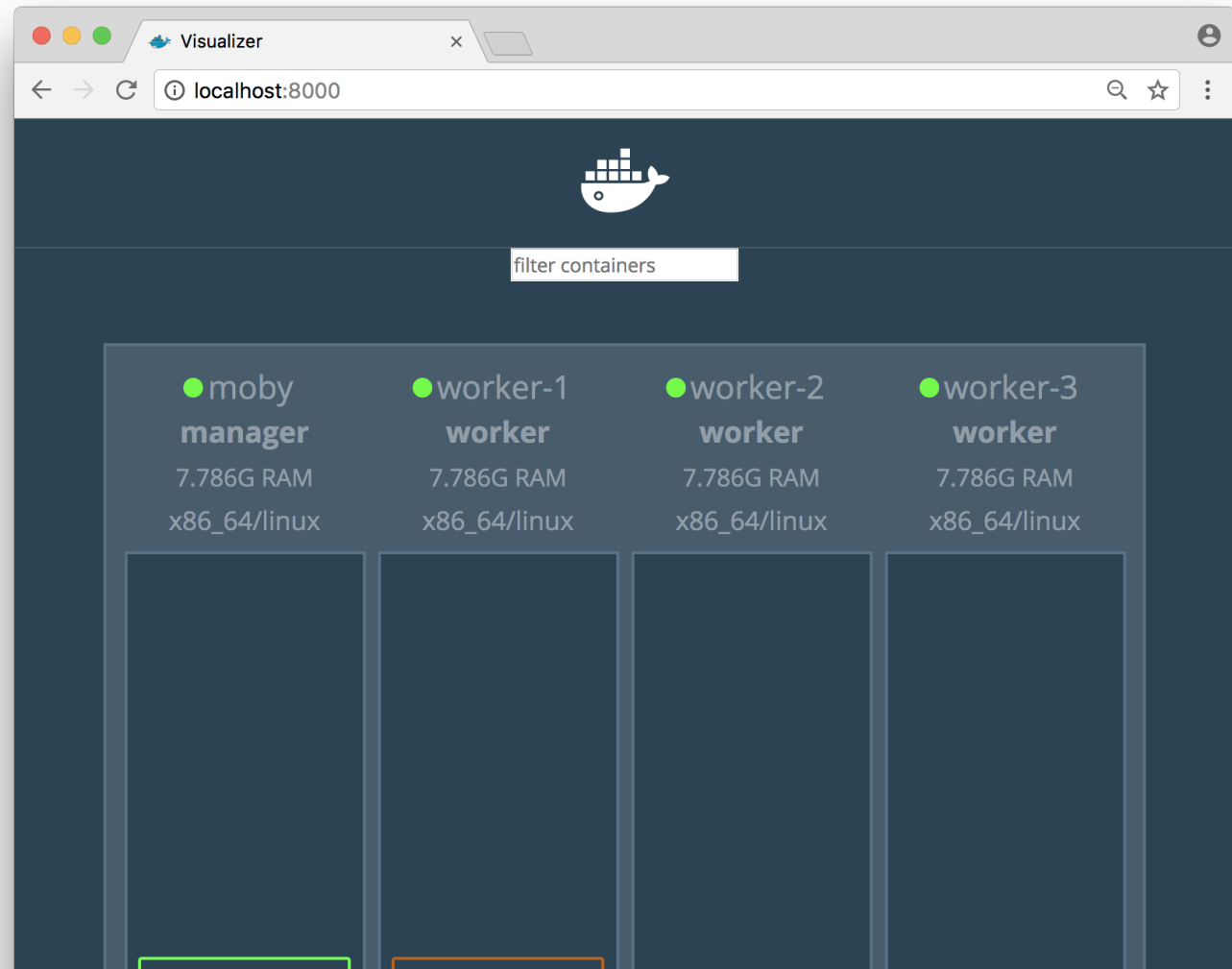
Now, let's deploy a service and try it out!

I have a very simple `quote-service` (written in Go) that creates some random quotes about successful programming languages. It can be deployed with the following command:

```
1 docker service create --name quotes-service --detach=false -p 8080:8080 magnuslars
```

Note: This time we set the `--detach` - flag to `false` meaning that the docker service create command waits to complete until the service is actually accessible, i.e. at least one container for the service reports to Docker that it is up and running.

Expect an output from the command like:



We can now try it out using `curl` :

```
1 curl localhost:8080/api/quote -s -w "\n" | jq .
```

Output should look like:

```
1 {
2   "ipAddress": "fc9193f817ef/",
3   "quote": "If I had to describe Go with one word it'd be 'sensible'.",
4   "language": "EN"
5 }
```

The most interesting part of the response from the service is actually the field `ipAddress` , that contains the hostname of the container that served the request, `fc9193f817ef` in the sample response above.

Scale

```
1 while true; do curl localhost:8080/api/quote -s -w "\n" | jq -r .ipAddress; sleep
```

Initially the output should return one and the same hostname, since we only have one container running in the service:

```
1 5fcdd3974d5c/  
2 5fcdd3974d5c/  
3 5fcdd3974d5c/  
4 5fcdd3974d5c/  
5 5fcdd3974d5c/  
6 5fcdd3974d5c/
```

Now, scale the `quote-service` by adding 10 new containers to it (11 in total):

```
1 docker service scale quotes-service=11 --detach=true
```



```
1 5aa06cf9a0df/  
2 8e02323f6b67/  
3 5fcdd3974d5c/  
4 b9f8da53ec30/  
5 be9489f011c3/  
6 559489f857d6/  
7 494d5b0a5b5a/  
8 4541532bc9bb/  
9 709a6b244149/  
10
```

Great, isn't it?

Resilience

Now, let's expose the container orchestrator, i.e. Docker in Swarm mode, to some problems and see if it handles them as expected!

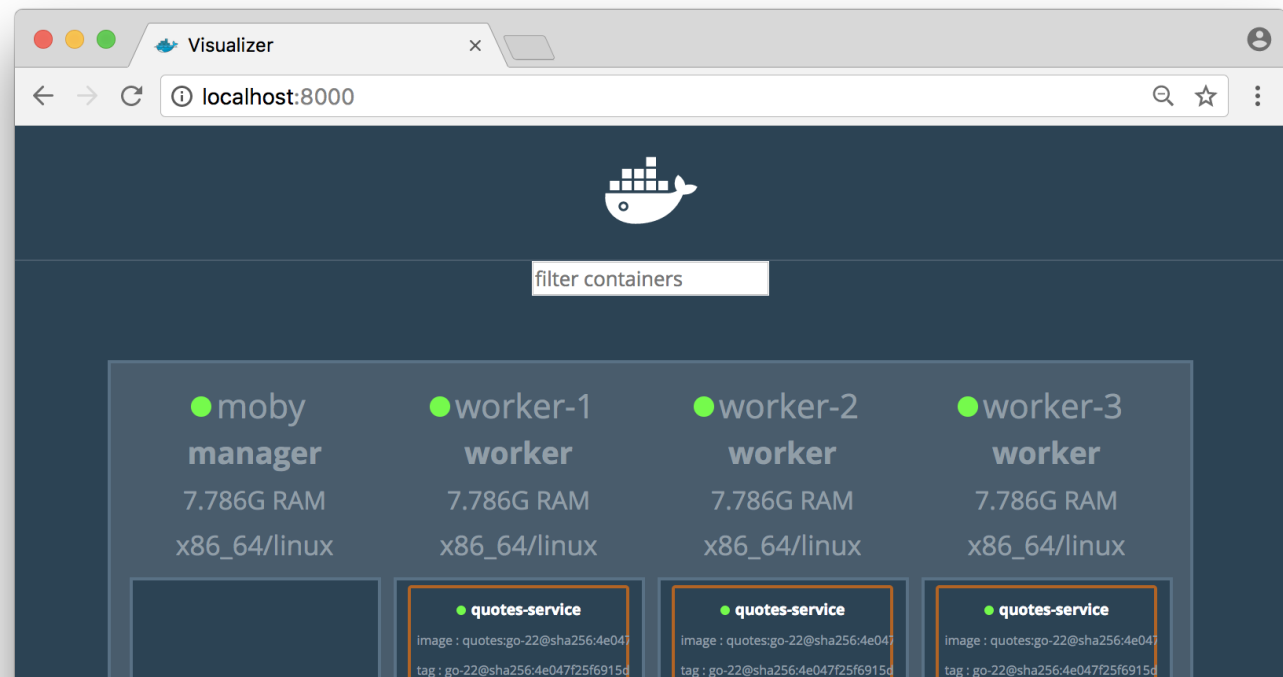
Kill Some Containers

First, let's shut down some arbitrary containers and see if the orchestrator detects it and start new ones!

To keep things relatively simple, let's kill all `note-services` running on the master node:


```
1 25ca7c327cb4
2 6d28d64ff831
```

Take a quick look (you have to be fast!) in the Visualizer and you will see that there are two missing `quote-service` containers in the master node:





But after a few seconds you will find the new containers started by the orchestrator to fulfill the desired state, i.e. 11 containers for the `quote-service` !

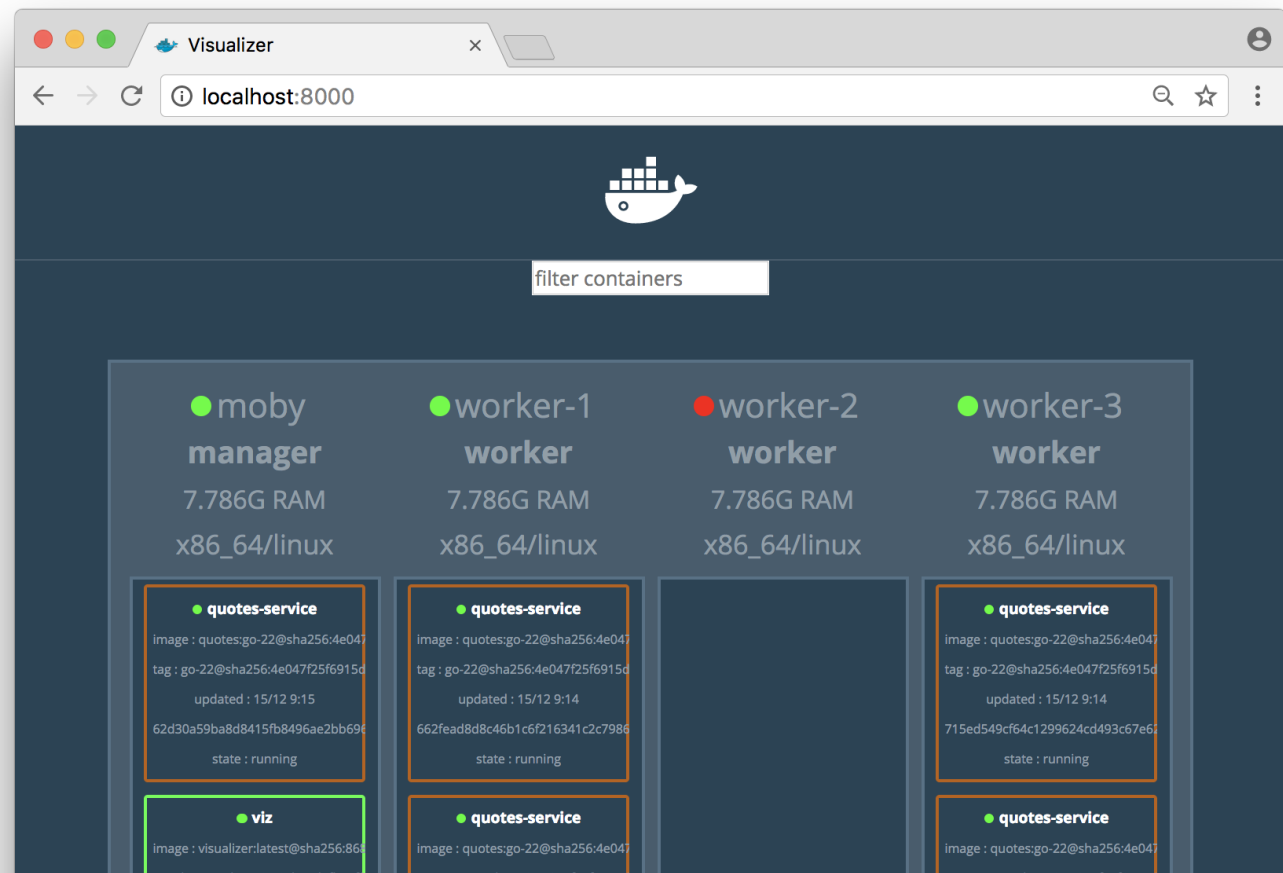
Crash a Worker Node

Now, let's make it even worse by killing a worker node, i.e. simulating that the node crash unexpected. Since we use Docker in Docker, this can be done by killing the worker nodes container. Let's shutdown `worker-2` :

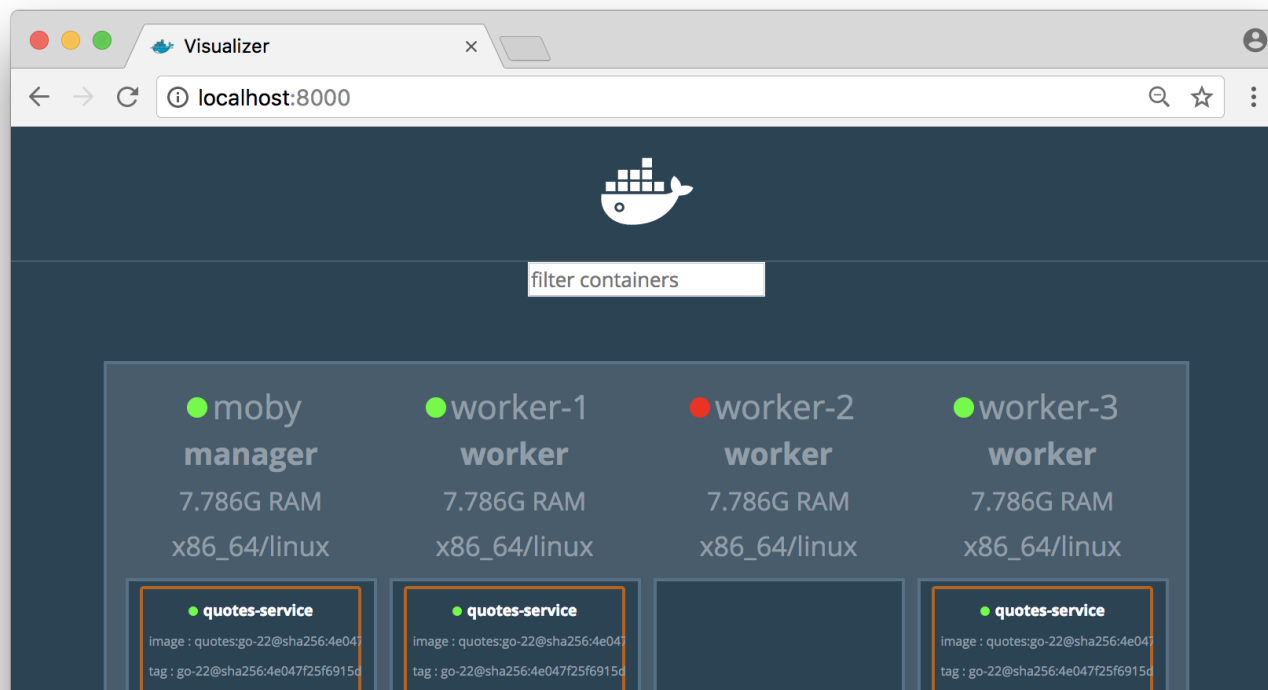
```
1  docker rm -f worker-2
```

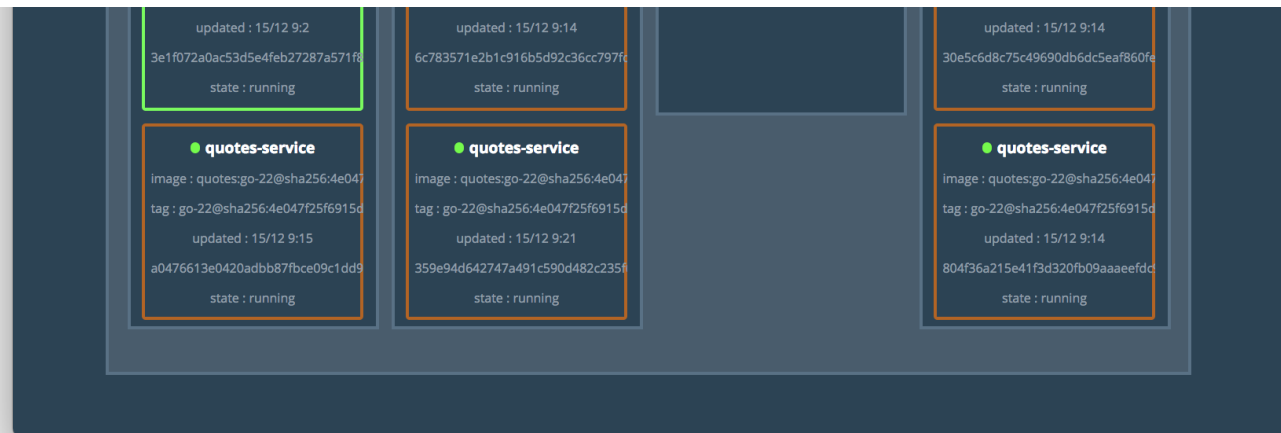
You will soon see how `worker-2` disappears from the `ctop` display:

...and the Visualizer also reports that `worker-2` is down:



...and soon after, the containers that was running on the failing node are rescheduled on the other nodes:





Let's wrap up with restarting the node again:

```

1  i=2
2  docker node rm worker-${i}
3  docker run -d --privileged --name worker-${i} --hostname=worker-${i} -p ${i}2375:2
4  docker --host=localhost:${i}2375 swarm join --token ${SWARM_TOKEN} ${SWARM_MASTER_

```

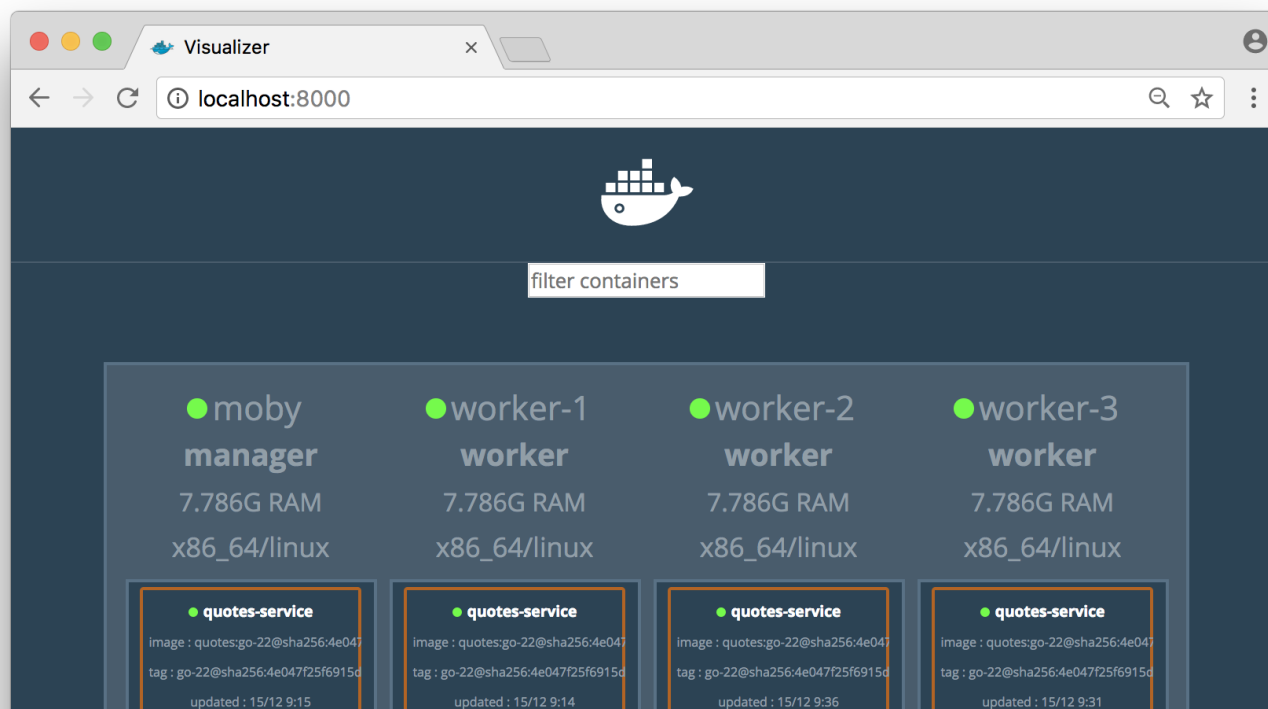
As you can see in the Visualizer, the new `worker-2` node is back online but the containers are not rebalanced back to the new worker node automatically:

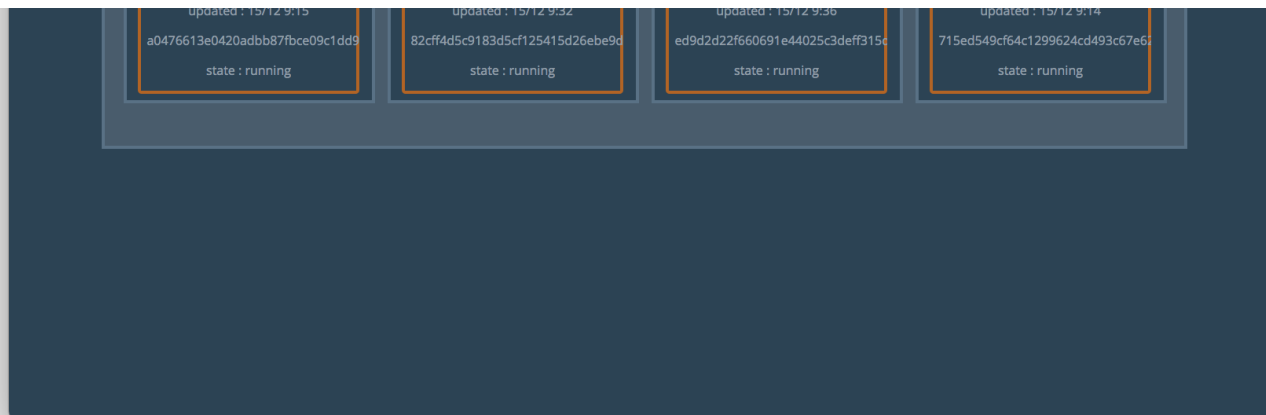
filter containers

● moby manager	● worker-1 worker	● worker-2 worker	● worker-3 worker
7.786G RAM x86_64/linux	7.786G RAM x86_64/linux	7.786G RAM x86_64/linux	7.786G RAM x86_64/linux
<ul style="list-style-type: none">● quotes-service image : quotes:go-22@sha256:4e047f25f6915d tag : go-22@sha256:4e047f25f6915d updated : 15/12 9:21 a1c7f468a894afeb0e69d566e1a768e state : running● quotes-service image : quotes:go-22@sha256:4e047f25f6915d tag : go-22@sha256:4e047f25f6915d updated : 15/12 9:15 62d30a59ba8d8415fb8496ae2bb69e state : running● viz image : visualizer:latest@sha256:86b83d5fb2aff tag : latest@sha256:86b83d5fb2aff updated : 15/12 9:2	<ul style="list-style-type: none">● quotes-service image : quotes:go-22@sha256:4e047f25f6915d tag : go-22@sha256:4e047f25f6915d updated : 15/12 9:14 662fead8d8c46b1c6f216341c2c7986 state : running● quotes-service image : quotes:go-22@sha256:4e047f25f6915d tag : go-22@sha256:4e047f25f6915d updated : 15/12 9:32 2872d084c1daad8da867053a208c6f state : running● quotes-service image : quotes:go-22@sha256:4e047f25f6915d tag : go-22@sha256:4e047f25f6915d updated : 15/12 9:21		<ul style="list-style-type: none">● quotes-service image : quotes:go-22@sha256:4e047f25f6915d tag : go-22@sha256:4e047f25f6915d updated : 15/12 9:31 0b80c7ac3bdeba3bf1cd7c8a88c03fe state : running● quotes-service image : quotes:go-22@sha256:4e047f25f6915d tag : go-22@sha256:4e047f25f6915d updated : 15/12 9:21 c182b79732d24d65e26693425403a state : running● quotes-service image : quotes:go-22@sha256:4e047f25f6915d tag : go-22@sha256:4e047f25f6915d updated : 15/12 9:14

You can, however, manually rebalance your containers with the commands:

```
1 docker service scale quotes-service=8 --detach=false
2 docker service scale quotes-service=11 --detach=false
```





Teardown

That's it, let's remove the Swarm cluster:

1. Remove the services
2. Unregister the worker nodes from the Swarm cluster
3. Shutdown the worker nodes (i.e. kill the worker node containers)
4. Remove the “Docker for Mac” Docker engine from the Swarm cluster


```
5   for i in $(seq "${NUM_WORKERS}"); do
6   docker --host=localhost:${i}2375 swarm leave
7   done
8
9   # Remove worker nodes
10  docker rm -f $(docker ps -a -q --filter ancestor=docker:${DOCKER_VERSION} --format
11
12  # Leave Swarm mode
13  docker swarm leave --force
```

Next up ...

Look out for the the the next part in the series where we will be setting up a Kubernetes cluster using Docker in Docker.

Leverage your CDN to optimize + secure your cloud infrastructure. Here's how.

Topics: DOCKER , DOCKER SWARM , MICROSERVICE ARCHITECTURE , DOCKER IN DOCKER , CLOUD , MULTINODE CONTAINER , CONTAINER ORCHESTRATION

Cloud Partner Resources

Use logic at the edge of the network — closer to your users — to strengthen and support your microservices
Fastly



Need to seamlessly go from deploying to scaling? Get the guide to gathering the right metrics.

DigitalOcean



Whitepaper: Site24x7: Powerful, Agile, Cost-Effective IT Management from the Cloud

Zoho



Three key metrics for monitoring AWS SNS performance and usage

Zoho

