Technology Conversations

Scaling To Infinity with Docker Swarm, Docker Compose and Consul (Part 4/4) – Scaling Individual Services

This series is split into following articles.

- A Taste of What Is To Come
- Manually Deploying Services
- Blue-Green Deployment, Automation and Self-Healing Procedure
- Scaling Individual Services

In the previous article we switched from manual to automatic deployment with Jenkins and Ansible. In the quest for **zero-downtime** we employed Consul to check **health** of our services and, if one of them fails, initiate deployment through Jenkins.

In this article we'll explore how to scale individual services.

Setup

For those of you who stopped VMs we created in the previous article (`vagrant halt`) or turned of your laptops, here's how to quickly get to the same state where we were before. The rest of you can skip this chapter.

```
vagrant up
vagrant ssh swarm-master
ansible-playbook /vagrant/ansible/infra.yml -i /vagrant/ansible/hosts/prod
ansible-playbook /vagrant/ansible/books-service.yml -i /vagrant/ansible/hosts/p
export DOCKER_HOST=tcp://0.0.0.0:2375
```

We can verify whether everything seems to be in order by running the following.

```
docker ps
curl http://10.100.199.200/api/v1/books | jq .
```

The first command should list, among other things, booksservice_[COLOR]_1 and booksservice_db_1 containers. The second one should retrieve JSON response with three books we inserted before.

With this out-of-the-way, we can continue where we left.

Scaling Services

Let us scale our **books-service** so that it is running on at least two nodes. That way we can be sure that if one of them fails, the other one will be running while the **rescue** setup we did in the previous article is finished and the failed service is redeployed.

```
docker ps | grep booksservice
cd /data/compose/config/books-service
docker-compose scale blue=2
docker ps | grep booksservice
```

If you are currently running **green**, please change the above command to `docker-compose scale green=2`.

The last `docker ps` command listed that two instances of our service are running; **booksservice_blue_1** and **booksservice_blue_2**.

As with everything else we did by now, we already added scaling option to the Ansible setup. Let's see how to do the equivalent of the command above with Ansible. We'll deploy the latest version of the service scaled to three instances.

```
ansible-playbook /vagrant/ansible/books-service.yml -i /vagrant/ansible/hosts/pı
docker ps | grep booksservice
```

With a single run of the **books-service** playbook, we deployed a new version of the service scaled to three instances.

We won't go into details but you can probably imagine the potential this has beyond simple scaling with the goal have one running when the other one fails. We could, for example, create a system that would scale services that are under heavy load. That can be done with Consul that could monitor services response times and, if they reach some threshold, scale them to meet the increased traffic demand.

Just as easy, we can scale down back to two services.

```
ansible-playbook /vagrant/ansible/books-service.yml -i /vagrant/ansible/hosts/pı
docker ps | grep booksservice
```

All this would be pointless if our **nginx** configuration would not support it. Even though we have multiple instances of the same service, **nginx** needs to know about it and perform load balancing across all of them. The Ansible playbook that we've been using already handles this scenario.

Let's take a look at the **nginx** configuration related to the **books-service**.

```
cat /data/nginx/includes/books-service.conf
```

The output is following.

```
location /api/v1/books {
  proxy_pass http://books-service/api/v1/books;
}
```

This tells **nginx** that whenever someone requests an address that starts with **/api/v1/books**, it should be proxied to **http://books-service/api/v1/books**. Let's take a look at the configuration for the **books-service** address (after all, it's not a real domain).

```
cat /data/nginx/upstreams/books-service.conf
docker ps | grep booksservice
```

The output will differ from case to case. The important part is that the list of nginx **upstream** servers should coincide with the list of services we obtained with `docker ps`. One possible output of the first command could be following.

```
upstream books-service {
    server  10.100.199.202:32770;
    server  10.100.199.203:32781;
}
```

This tells **nginx** to balance requests between those two servers and ports.

We already mentioned in the previous articles that we are creating nginx configurations using Consul Template. Let us go through it again. The **blue** template looks like this.

```
upstream books-service {
    {{range service "books-service-blue" "any" }}
    server {{.Address}}:{{.Port}};
    {{end}}
}
```

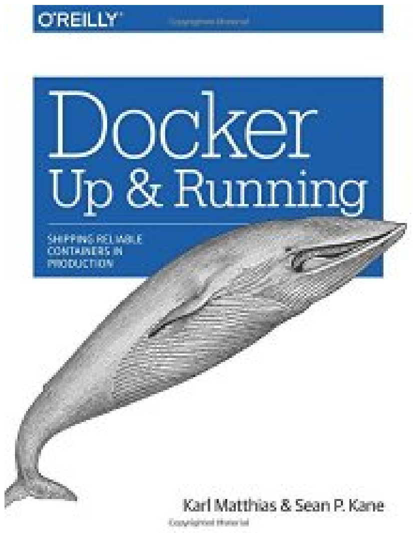
It tells Consul to retrieve all instances (range) of the service called **books-service-blue** ignoring their status (any). For each of those instances it should write the **IP** (.Address) and **port** (.Port). We created a template for both **blue** and **green** versions. When we run the last deployment, Ansible took care of creating this template (with correct color), copying it to the server and running **Consul Template** which, in turn, reloaded **nginx** at the end of the process.

The current setting does not scale MongoDB. I'll leave that up to you. The process should be the same as with the service itself with additional caveat that Mongo should be set to use Replica Set with one instance set as primary and the rest as secondary instances.

The End (For Now)

We covered a lot of ground in these four articles and left even more possibilities unexplored. We could, for example, host on the same cluster not only different services but also copies of the same services for multiple customers. We could create logic that deploys services not only to nodes that have the least number of containers but those that have enough CPU or memory. We could add

Kubernetes or Mesos to the setup and have more powerful and precise ways to schedule deployments. We already had CPU, memory and HD checks set in Consul but no action is taken when they reach their thresholds. However, time is limited and not all can be explored at once.


O'REILLY
Docker
Up & Running
SHIPPING RELIABLE CONTAINERS IN PRODUCTION
Karl Matthias & Sean P. Kane

I'd like to hear from you what would be the next subject to explore, which parts of these articles require more details or your experience after trying to apply this in your organization.

If you have any trouble following these examples, please let me know and I'll give my best to help you out.

This entry was posted in Continuous Integration, Delivery and Deployment, Docker, Tutorial and tagged Ansible, Consul, Continuous delivery, Continuous integration, Continuous Integration, Delivery and Deployment, Data Center, Docker, Docker Compose, Docker Swarm, jenkins, Microservices, nginx, Scaling, Server Farm, Vagrant on July 2, 2015 [http://technologyconversations.com/2015/07/02/scaling-to-infinity-with-docker-swarm-docker-compose-and-consul-part-44-scaling-individual-services/] by Viktor Farcic.

2 thoughts on "Scaling To Infinity with Docker Swarm, Docker Compose and Consul (Part 4/4) – Scaling Individual Services"

Florian
August 17, 2015 at 10:57 pm

Hi Viktor, thank you so much for this great series of posts – it was a pleasure reading them and exploring further opportunities.

Just one little question regarding the first scaling operation: You said "If you are currently running green, please change the above command to docker-compose scale blue=2" – am I right this should rather be "… to docker-compose scale green=2" then?

All the best from Berlin,
Florian

Viktor Farcic Post author
August 17, 2015 at 11:06 pm

Hi Florian,

You’re right. Thank you for commenting it. I updated the article