Technology Conversations

Scaling To Infinity with Docker Swarm, Docker Compose and Consul (Part 2/4) – Manually Deploying 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

The previous article showed how scaling across the server farm looks like. We'll continue where we left and explore details behind the presented implementation. Orchestration has been done through Ansible. Besides details behind tasks in Ansible playbooks, we'll see how the same result could be accomplished using manual commands in case you might prefer a different orchestration/deployment framework.

We won't go into details how to set up Consul, Docker Compose, Docker Swarm, nginx, etc. They can be seen by looking at the Ansible playbooks in the vfarcic/docker-swarm GitHub repository.

Creating New Servers

For the sake of a better explanation, if you followed the previous article, please destroy your VMs. We'll start over and explain each task one by one.

```
vagrant destroy
```

Let's create our virtual machines and setup infrastructure using few Ansible playbooks. If you are asked whether you want to continue connecting, please answer with **yes**.s

```
vagrant up
vagrant ssh swarm-master
ansible-playbook /vagrant/ansible/swarm.yml -i /vagrant/ansible/hosts/prod
ansible-playbook /vagrant/ansible/compose.yml -i /vagrant/ansible/hosts/prod
ansible-playbook /vagrant/ansible/nginx.yml -i /vagrant/ansible/hosts/prod
ansible-playbook /vagrant/ansible/consul.yml -i /vagrant/ansible/hosts/prod
```

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

```
export DOCKER_HOST=tcp://0.0.0.0:2375
docker info
docker ps -a
```

The second command should show that there are three nodes in the cluster. The third should list 9 containers; swarm-node, registrator and registrator-kv on each of the three nodes we have.

Now it's time to start working on deployments.

Even though the previous article had all the commands defined within Ansible playbooks, we'll do all the steps manually so that you can get an understanding what's going on.

Service Deployment

We'll use **Docker Compose** to run our containers. It has a very simple syntax based on YML. Those familiar with Ansible will feel familiar with it.

In all previous articles we used Ansible for this task. My opinion was that Ansible offers everything that Docker Compose does and so much more. Docker compose is only concerned with building, running and other Docker operations. Ansible is meant to orchestrate everything, from server setup, deployments, building, etc. It is one tool that can take care of all orchestration and deployment steps.

However, Ansible Docker module does not work well with Swarm. Once we're finished with manual commands, we'll continue using Ansible for all the tasks except running Docker containers through Swarm.

We'll be deploying **books-service**. It is an application that provides REST APIs to list, update or delete books. Data is stored in a Mongo database.

Setup Docker Compose Files on the Server

First step is to set up Docker Compose templates. We'll need a directory where those templates will reside and a template itself.

Creating directory is easy.

```
sudo mkdir -p /data/compose/config/books-service
```

Creating Docker Swarm template, is a bit harder in our case. Since we're building truly distributed applications, we don't have all the information in advance. The service we'll be deploying needs a link to another container hosting Mongo DB. That container can end up being deployed to any of the three servers we just brought up.

What we want to accomplish is something similar to the following Docker Compose configuration.

```
ports:
  - 8080
environment:
  - SERVICE_NAME=books-service
  - DB_PORT_27017_TCP=[IP_AND_PORT_OF_THE_MONGO_DB_SERVICE]
image: vfarcic/books-service
```

We want to expose internal port 8080 (that's the one service is using). For the outside world, Docker will map that port to any port it has available. We'll name the service **books-service**.

Now comes the tricky part, we need to find out what the DB **IP** and **port** are before we create this template.

In order to solve this problem, we'll create Consul template instead. Run the following command.

```
cd /data/compose/config/books-service
echo 'db:
  ports:
    - 27017
  environment:
    - SERVICE_ID=books-service-db
  image: mongo
blue:
  ports:
    - 8080
  environment:
    - SERVICE_NAME=books-service-blue
    - DB_PORT_27017_TCP={{ key "services/mongo/books-service-db" }}

  image: vfarcic/books-service
green:
  ports:
    - 8080
  environment:
    - SERVICE_NAME=books-service-green
    - DB_PORT_27017_TCP={{ key "services/mongo/books-service-db" }}

  image: vfarcic/books-service
' | sudo tee docker-compose.yml.ctmpl
sudo cp docker-compose.yml.ctmpl docker-compose.yml
```

We created a new template /data/compose/config/books-service/docker-compose.yml.ctmpl. "Strange" things inside `{%` and `%}` will be explained soon. For now, it suffices to say that value of the **DB_PORT_27017_TCP** will be replaced by **books-service-db** IP and port.

Let's go through the template quickly. First we're defining **db** container that exposes port 27017 (standard Mongo port), sets environment variable SERVICE_ID (we'll use it later) and specifies that the image is **mongo**. Similar is done for the **books-service** except that we're specifying it twice. Once as **blue** and the other one as **green**. We'll be practicing blue/green deployment in order to accomplish **no downtime** goal (more info can be found in Continuous Deployment Strategies article).

We could have made Mongo DB always run on the same server as the books-service but that would cause potential problems. First, it would mean that all three containers (db, blue and green) need to be on the same server. While that might be OK in this relatively small example, on big systems this would create a bottleneck. More freedom we have to distribute containers, more CPU, memory and HD utilization we'll squeeze out of our servers.

Run the DB Container

Running DB container is easy since it does not depend on any other service. We can simply run the **db** target we specified earlier.

```
docker-compose pull db
docker-compose up -d --no-recreate db
```

The first command pulled **db** to all the nodes (we'll get into reasons behind this command a bit later).

Second command argument **up** tells compose that we'd like him to make sure that it is up and running, `-d` means that it should run in detached mode, `--no-recreate` tells compose not do do anything if container is already running and, finally, last argument is the name we specified in the **docker-compose.yml**.

Let's see where was it deployed.

```
docker ps | grep booksservice_db
```

You'll see the IP and the port of the **db** service.

Run the Service Container For the First Time

Running the service container will be a bit more complicated. There are few obstacles that we didn't face with the database. The major one is that we need to know the IP and the port of the database we just deployed and pass that information. Later on when we run the service for the second time (new release), things will get even more complicated.

At the moment, our major problem is to find out the IP and the port of the database service we just deployed. This is where **Consul Template** comes in handy.

Before we run the command, let us see how does the environments section of the Consul template **docker-compose.yml.ctmpl** looks like.

```
cat docker-compose.yml.ctmpl | grep DB_PORT_27017_TCP
```

The output should be following.

```
- DB_PORT_27017_TCP={{ key "services/mongo/books-service-db" }}
- DB_PORT_27017_TCP={{ key "services/mongo/books-service-db" }}
```

Now let us run Consul Template.

```
sudo consul-template -consul localhost:8500 -template "docker-compose.yml.ctmpl
```

Let's take a look at the **docker-compose.yml** that was just created.

```
cat docker-compose.yml | grep DB_PORT_27017_TCP
```

The result might be different depending on the location (IP and port) Docker Swarm choose for our DB service. In my case, the output is the following.

```
- DB_PORT_27017_TCP=10.100.199.202:32768
- DB_PORT_27017_TCP=10.100.199.202:32768
```

Consul Template put the correct database IP and port. How did this happen? Let's us first go through the command arguments.

- **-consul** let's us specify the address of our Consul instance (localhost:5000).
- **-template** consist of two parts; source and destination. In this case we're telling it to use docker-compose.yml.ctmpl as template and product docker-compose.yml as output.
- **-once** is self-explanatory. This should run only one time.

The real "magic" is inside the template. We have the following line in docker-compose.yml.ctmpl.

```
{{ key "services/mongo/books-service-db" }}
```

This tells Consul Template to look for a key **services/mongo/books-service-db** and replace this with its value.

We can have a look at the value of that key using the following command.

```
curl http://localhost:8500/v1/kv/services/mongo/books-service-db?raw
```

The only mystery left unsolved is how this information got to Consul in the first place. The answer is in a handy tool called registrator. It allows us to monitor containers and update Consul key/value store whenever one is run or stopped. We already set it up with Ansible so when we run the database service, it detected a new container and updated Consul accordingly.

Now that we have our docker-compose.yml correctly updated with the database information, it is time to pull the latest release of our service.

```
docker-compose pull blue
```

This command pulled the latest release of our application to all of the servers in the cluster. While we could have limited it only to the server we'll be running on, having it on all of them helps reacting swiftly in case of a problem. For example, if one node goes down, we can run the same release anywhere else very fast since we won't be wasting time in pulling the image from the registry.

Now we can run the container.

```
docker-compose up -d blue
docker ps | grep booksservice_blue
```

The second command listed the newly run service (blue). Among other things, you can see the IP and port it is running on.

For our future convenience, we should tell Consul that we just deployed **blue** version of our service.

```
curl -X PUT -d 'blue' http://localhost:8500/v1/kv/services/books-service/color
```

We're still not done. Even though the application is up and running and correctly pointing to the database running on a different server, we still did not solve the port problem. Our service should be accessible from http://10.100.199.200/api/v1/books and not one of the servers Swarm deployed it to. Also, we should be able to use it through the port 80 (standard http) and not a random port that was assigned to us. This can be solved with nginx reverse proxy and Consul Template. We can update nginx configuration in a similar way as we updated docker-compose.yml.

First we'll create few nginx configuration files.

```
echo '
server {
    listen 80;
    server_name 10.100.199.200;
    include includes/*.conf;
}' | sudo tee /data/nginx/servers/common.conf

echo '
location /api/v1/books {
  proxy_pass http://books-service/api/v1/books;
}' | sudo tee /data/nginx/includes/books-service.conf
```

We'll also need two more Consul templates.

```
echo '
upstream books-service {
    {{range service "books-service-blue" "any" }}
    server {{.Address}}:{{.Port}};
    {{end}}
}
' | sudo tee /data/nginx/templates/books-service-blue-upstream.conf.ctmpl
echo '
upstream books-service {
    {{range service "books-service-green" "any" }}
    server {{.Address}}:{{.Port}};
    {{end}}
}
' | sudo tee /data/nginx/templates/books-service-green-upstream.conf.ctmpl
```

This template is a bit more complicated. It tells Consul to retrieve all instances of a service (range) 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.

At the moment this setting might be more complicated than we need since we're running only one instance of a service. Later on we'll go deeper and see how to scale not only difference services but also the same service across multiple servers.

Let's apply the blue template.

```
sudo consul-template -consul localhost:8500 -template "/data/nginx/templates/bo
cat /data/nginx/upstreams/books-service.conf
```

The only new thing here is the third argument in **-template**. After specify the source and the destination, we're telling it to restart nginx by running `docker kill -s HUP nginx` command.

The output of the newly created file would be similar to the following.

```
upstream books-service {
    server 10.100.199.203:32769;
}
```

Finally, let us test whether everything works as expected.

```
curl -H 'Content-Type: application/json' -X PUT -d '{"_id": 1, "title": "My Fir
curl -H 'Content-Type: application/json' -X PUT -d '{"_id": 2, "title": "My Sec
curl -H 'Content-Type: application/json' -X PUT -d '{"_id": 3, "title": "My Thi
curl http://10.100.199.200/api/v1/books | jq .
```

The last curl command should output three books that we inserted previously.

```
[
  {
    "_id": 1,
    "title": "My First Book",
    "author": "Joh Doe"
  },
  {
    "_id": 2,
    "title": "My Second Book",
    "author": "John Doe"
  },
  {
    "_id": 3,
    "title": "My Third Book",
    "author": "John Doe"
  }
]
```

To Be Continued

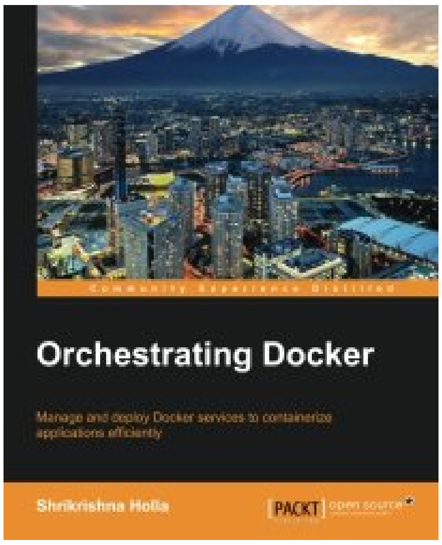
We managed to manually deploy one database and one REST API service. Both of them were not deployed to a server we specified in advance but to the one that had the least number of containers running.

We still have a lot of ground to cover. The next release of our service should do a few more steps that we did not do yet. Without those additional steps we would not have blue/green deployment and there would be some downtime every time we release a new version.

There are additional benefits we can squeeze from Consul like health checking that will, together with Jenkins, redeploy our services whenever something goes wrong.

Further more, we might want to have an option not only to scale different services but also to scale the same service across multiple servers.

Finally, everything we did by now was manual and we should use Ansible playbooks that will do all those things for us.


Community Experience Distilled
Orchestrating Docker
Manage and deploy Docker services to containerize applications efficiently
Shrikrishna Holla
PACKT open source

The story continues in the Blue-Green Deployment, Automation and Self-Healing Procedure article.

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-24-manually-deploying-services/] by Viktor Farcic.

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

Pingback: pinboard August 11, 2015 — arghh.net

ignasi35
October 20, 2015 at 9:02 pm

Mother of mercy! My swarm has two extra machines from external IPs (I suspect it's linode machines given the kernelversion)

```
vagrant@swarm-master:~$ docker info
Containers: 22
Images: 22
Role: primary
Strategy: spread
```

```
Filters: affinity, health, constraint, port, dependency
Nodes: 5
localhost: 104.237.138.145:2375
 └ Containers: 4
 └ Reserved CPUs: 0 / 2
 └ Reserved Memory: 0 B / 2.05 GiB
 └ Labels: executiondriver=native-0.2, kernelversion=4.1.5-x86_64-linode61,
operatingsystem=Ubuntu 14.04.1 LTS, storagedriver=devicemapper
localhost: 104.237.134.7:2375
 └ Containers: 9
 └ Reserved CPUs: 0 / 2
 └ Reserved Memory: 0 B / 2.05 GiB
 └ Labels: executiondriver=native-0.2, kernelversion=4.1.5-x86_64-linode61,
operatingsystem=Ubuntu 14.04.1 LTS, storagedriver=devicemapper
swarm-node-01: 10.100.199.201:2375
 └ Containers: 3
 └ Reserved CPUs: 0 / 1
 └ Reserved Memory: 0 B / 1.019 GiB
 └ Labels: executiondriver=native-0.2, kernelversion=3.13.0-66-generic, operatingsystem=Ubuntu
14.04.3 LTS, storagedriver=devicemapper
swarm-node-02: 10.100.199.202:2375
 └ Containers: 3
 └ Reserved CPUs: 0 / 1
 └ Reserved Memory: 0 B / 1.019 GiB
 └ Labels: executiondriver=native-0.2, kernelversion=3.13.0-66-generic, operatingsystem=Ubuntu
14.04.3 LTS, storagedriver=devicemapper
swarm-node-03: 10.100.199.203:2375
 └ Containers: 3
 └ Reserved CPUs: 0 / 1
 └ Reserved Memory: 0 B / 1.019 GiB
 └ Labels: executiondriver=native-0.2, kernelversion=3.13.0-66-generic, operatingsystem=Ubuntu
14.04.3 LTS, storagedriver=devicemapper
CPUs: 7
Total Memory: 7.157 GiB
Name: 619d7c527220
```

How could that happen?
I'm on a newly installed machine. I've got fresh VB, vagrant, etc… only reused item are my ssh keys.

Any idea what's going on?
I've restarted the Vagrant VMs (I haven't rebuild the VMs with a destroy) and the swarm is still made of 5 nodes+master.

I've followed this post in my old laptop and can't reproduce this situation. I'm completely puzzled.

Viktor Farcic Post author
October 20, 2015 at 11:17 pm

You just discovered how easy it is to add more machines to an existing Swarm cluster . If you open the ansible/roles/swarm/defaults/main.yml file you'll see the swarm_cluster_id variable. Swarm uses it to identify all the nodes in the cluster. Every machine with the Swarm within the same network and the same Swarm ID automatically becomes a member. You can obtain a new ID by running the "swarm create" command (when run inside a container it should be "docker run -it –rm swarm swarm create" if I remember it from memory). You can find more info in https://docs.docker.com/swarm/discovery/. In this article we're using the hosted discovery. In a "real world" situation we should probably choose some service discovery tool like Consul or etcd.

Anyways, what happened to you means that someone else in your network is trying the same article . Just change the ID and you'll be alone again.

ignasi35
October 20, 2015 at 10:20 pm

Oh, dear. I destroyed the VMs and rebuilt them from scratch and it happened again.

ken
October 23, 2015 at 1:13 am

Very nice series of articles! Can you explain how this key "services/mongo/books-service-db" populated?

ken
October 23, 2015 at 1:15 am

umm… found my answer:
"The only mystery left unsolved is how this information got to Consul in the first place. The answer is in a handy tool called registrator. It allows us to monitor containers and update Consul key/value store whenever one is run or stopped. We already set it up with Ansible so when we run the database service, it detected a new container and updated Consul accordingly."

Viktor Farcic Post author
October 23, 2015 at 1:33 pm

I'm glad you found the answer.