

Technology Conversations

Scaling To Infinity with Docker Swarm, Docker Compose and Consul (Part 3/4) – Blue-Green Deployment, Automation and Self-Healing Procedure

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 manually deployed the first version of our service together with a separate instance of the Mongo DB container. Both are (probably) running on different servers. **Docker Swarm** decided where to run our containers and **Consul** stored information about service IPs and ports as well as other useful information. That data was used to link one service with another as well as to provide information nginx needed to create proxy.

We'll continue where we left and deploy a second version of our service. Since we're practicing blue/green deployment, the first version was called **blue** and the next one will be **green**. This time there will be some additional complications. Deploying the second time is a bit more complicated since there are additional things to consider, especially since our goal is to have no downtime.

Setup

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

```
1 vagrant up
2 vagrant ssh swarm-master
3 ansible-playbook /vagrant/ansible/infra.yml -i /vagrant/ansible/hosts/prod
4 export DOCKER_HOST=tcp://0.0.0.0:2375
5 docker start booksservice_db_1
6 docker start booksservice_blue_1
7 sudo consul-template -consul localhost:8500 -template "/data/nginx/templates/bo
```

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

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

The first command should list, among other things, booksservice_blue_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.

Run the Second Release Of The Service Container

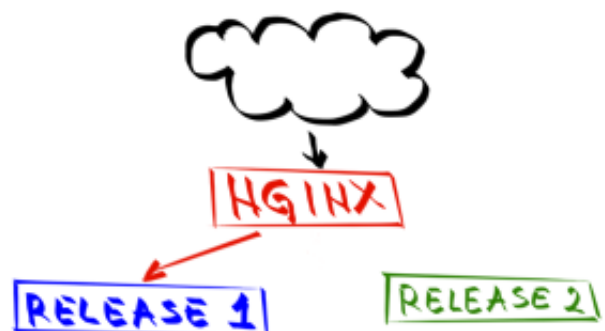
At this moment we have only **blue** release deployed and accessible to our users through **nginx**.



The next release will be called **green**. We'll run it in parallel with the previous one in order to avoid any downtime. During this time, all our users will continue being redirected to the old (**blue**) release. Once everything is up and running and we're sure that the new release (**green**) is working as expected, we'll reconfigure **nginx** to proxy requests to the new (**green**) release and the old one (**blue**) will be stopped.

```
1 | cd /data/compose/config/books-service
2 | docker-compose pull green
3 | docker-compose rm -f green
4 | docker-compose up -d green
5 | docker ps | grep booksservice
```

At this moment we have two versions of our service up and running (old and new; blue and green). That can be seen from the output of the last command (`docker ps`). It should display both services running on different servers (or the same if that turned up to be the place with the least number of containers).



This is the moment when we should run our automated tests. We do not have them prepared for this article so we'll manually execute `curl` command to check whether everything seems to be OK.

```
1 | curl http://localhost:8500/v1/catalog/service/books-service-green | jq .
2 | curl http://[IP]:[PORT]/api/v1/books | jq .
```

The first command queries Consul and returns data related to the books-service-green service. Please make sure to change the [IP] and the [PORT] in the second command (`curl`) to values you got

from Consul.

Keep in mind that we tested the service by sending request directly to its IP and port and not to the publicly available address served with **nginx**. At this moment both services are running with the old one (blue) being available for the public on <http://10.100.199.200> and the new one (green) at the moment accessible only to us. Soon we'll change nginx and tell it to redirect all requests to the new one thus archiving zero-downtime. Now that everything seems to be working as expected, it is time to make the new release (green) available to the public.

```
1 curl -X PUT -d 'green' http://localhost:8500/v1/kv/services/books-service/color
2 sudo consul-template -consul localhost:8500 -template "/data/nginx/templates/bo
3 docker stop booksservice_blue_1
4 curl http://10.100.199.200/api/v1/books | jq .
5 docker ps -a | grep booksservice
```

First we put a new value (green) to the **books-service/color** Consul key. That is for the future reference in case we want to know what color is currently running. This information will come in handy when we reach the point of having all this fully automated. Then we updated **nginx** configuration by running **consul-template**.

Next, `curl` command was the final piece of testing that verifies that publicly available

service (this time green) continues working as expected and with no down-time. This testing should be automated but, for the sake of brevity, testing automation was skipped from this article. Finally, we're checking the output of the `docker ps -a` command. It should show `booksservice_green_1` as **Up** and `booksservice_blue_1` as **Exited**.



Automation With Ansible And Jenkins

Everything we did by now was fine as a learning exercise but in the “real world” all this should be automated. We'll use Ansible as orchestration tool to run all the commands we did by now and a few more. We won't go into details of the Ansible playbook `books-service.yml`. It can be found together with the rest of source code in the [docker-swarm](#) GitHub repository. Since Ansible playbook follows the same logic as manual command we run and, in general, its playbooks are very easy to read, hopefully you won't have a problem understanding it without further explanation. If you run into problems, please consult [Continuous Integration, Delivery and Deployment](#). It has quite a few articles dedicated to Ansible. Feel free to send comments (below) or contact me directly with questions if there are any.

Important thing to note is that there are no separate roles for different services. When Ansible is used with micro services architecture, having a separate role for each service can quickly become too much to manage. Since with Docker it is easy to standardize deployments, there is only one role called **service**. That role uses variables to customize few differences between different micro

services. For example, one might use a database (for example **books-service**) while other doesn't (for example **books-fe**). In other words, each service has a separate playbook with service specific variables and all of them rely on the same role.

As an example, following is the definition of the **books-service.yml** playbook.

```

1  - hosts: service
2      remote_user: vagrant
3      sudo: yes
4      vars:
5          - container_image: books-service
6          - container_name: books-service
7          - http_address: /api/v1/books
8          - has_db: true
9      roles:
10         - docker
11         - consul
12         - swarm
13         - nginx
14         - service

```

We have service specific variables defined under **vars** section followed with all the **roles** required for this service. Variables are defining the image and the name of the Docker container, HTTP address service should run on (used to provide nginx proxy) and whether it has a database or not. All services depend on the same roles. They need **Docker**, **Consul**, **Swarm**, **nginx** and **service**. All but the last one are dependencies. The **service** role defines this and all other services we're running. If you take a look at the **books-fe.yml** playbook, you'll notice that the only difference between the two are variables. Please take a look at the source code in the [docker-swarm](#) GitHub repository for more info.

Now, let us run **books-service.yml** playbook to deploy yet another version of our service.

```
1  ansible-playbook /vagrant/ansible/books-service.yml -i /vagrant/ansible/hosts/p
```

Did was quite a better experience than what we did by now. A single command took care of everything.

Did it work correctly? We can check that in the same was as before.

```

1  curl http://localhost:8500/v1/kv/services/books-service/color?raw
2  curl http://10.100.199.200/api/v1/books | jq .
3  docker ps -a | grep booksservice

```

We can see that the current color is blue (previous was green), that nginx works correctly, that the old (green) container is stopped and that the new one (blue) is running.







Can we do it without even running that single command? That's what Jenkins (and similar tools) are for. If you already used Jenkins you might think that you should skip this part. Please bear with me because once we set it up, we'll go through ways to recuperate from failure.

Let us first make sure that Jenkins is up and running.

```
1 | ansible-playbook /vagrant/ansible/jenkins.yml -i /vagrant/ansible/hosts/prod
```



Great thing about orchestration tools like Ansible is that they always check the state and act only when needed. If Jenkins was already up-and-running, Ansible did nothing. If, on the other hand, it was not deployed or was shut down, Ansible acted accordingly and made it work again.

Jenkins can be viewed by opening <http://10.100.199.200:8080/> in your favourite browser. Among other jobs, you should see the **books-service** job that we'll use to deploy new version of our service.

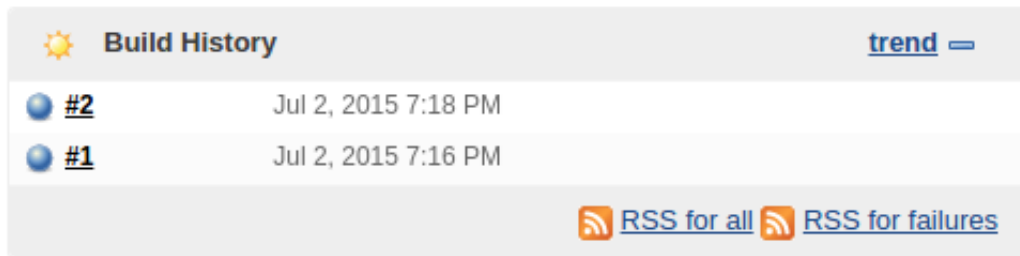
All +						
S	W	Name ↓	Last Success	Last Failure	Last Duration	
		books-fe	N/A	N/A	N/A	
		books-service	N/A	N/A	N/A	

Next, create a new node by using following steps.

- Click **Manage Jenkins > Manage Nodes > New Node**
- Name it **cd**, select **Dumb Slave** and click **OK**
- Type **/data/jenkins/slaves/cd** as **Remote Root Directory**
- Type **10.100.199.200** as **Host**
- Select **Launch slave agents on Unix machines via SSH**
- Click **Add* next to **Credentials**
- Use **vagrant** as both **Username** and **Password** and click **Add**
- Click **Save**

S	Name ↓	Architecture	Clock Difference	Free Disk Space
	cd	Linux (amd64)	In sync	22.03 GB
	master	Linux (amd64)	In sync	22.03 GB
Data obtained		7.7 sec	7.7 sec	7.7 sec

Now we can run the **books-service** job. From the Jenkins home page, click **books-service** link and then the **Build Now** button. Refresh the page and you'll see the icon in **Build History** blinking until it's finished running. Once done, there will be a blue icon indicating that everything run correctly.



Run `docker ps` to confirm that the service is running.

```
1 | docker ps -a | grep booksservice
```

What is missing from this Jenkins job configuration is **Source Code Management** setup that would pull the code from repository whenever there is any change and initiate deployment. After all, we don't want to waste time pushing the **Build** button every time someone changes the code. Since this is being run in local, SCM setup cannot be demonstrated. However, you should have not trouble finding the information online.

Now let us move to more interesting things and see how to recuperate from failures.

Self-Healing System

What happens when something goes wrong? What happens when, for example, one service or the whole node goes down? Our system should be able to recuperate from such problems. **Self-Healing Systems** is a big topic spanning code architecture, servers setup, notifications, etc. This subject deserves at least a whole article (and probably a whole book) so we won't go deep into **Self-Healing** concepts but only take the simplest scenario. We'll set the system in a way that when one service goes down it is redeployed. If, on the other hand, the whole node stops working, all containers from that node will be transferred to a healthy one. We'll need Consul, Jenkins and Ansible for that.

Here's the flow we want to accomplish. We need something to monitor **health** of our services. We'll use **Consul** for this purpose. If one of them does not respond to a request made every 10 seconds, Jenkins job will be called. Jenkins, in turn, will run **Ansible** that will make sure that everything related to that service is up and running. Since we're using **Docker Swarm**, if the cause of service not working is server shutdown, healthy node will be selected instead. We could have done this without Jenkins but since we already have it set up and it provides a lot of nice and easy to configure features (even though we don't use them in this example), we'll stick with it.

Before we go into details, let's see it in action. We'll start by stopping our service.

```
1 | docker ps | grep booksservice
2 | docker stop booksservice_green_1
3 | docker stop booksservice_blue_1
4 | docker ps -a
```

At this moment I could not be sure whether you are running **blue** or **green** version so the

commands above will try to stop both. The second `docker ps -a` command is used to verify that the service is indeed stopped (status should be **Exited**).

Now that we stopped our service, Consul will detect that since it runs verification every 10 seconds. We can see its log with the following.

```
1 | cat /data/consul/logs/watchers.log
```

The output should be similar to the one below.

```
1 | Consul watch request:
2 | [{"Node": "swarm-master", "CheckID": "service:books-service", "Name": "Service 'book:
3 |
4 | >>> Service books-service is critical
5 |
6 | Triggering Jenkins job http://10.100.199.200:8080/job/books-service/build
```

If you don't see the same output, please make sure that 10 seconds passed between stopping the service and checking Consul logs.

Consul detected that there was something wrong and made a request that triggered the same Jenkins job we run manually and performed another deployment.

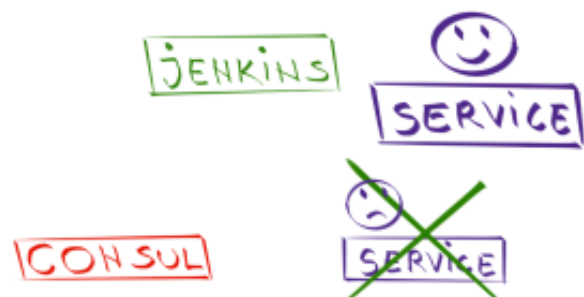
You can see the deployment progress from Jenkins. Once it is finished we can take another look at containers status.

```
1 | docker ps | grep booksservice
```

Assuming that you stopped the **green** version, this time you would see **booksservice_blue_1** running.

Same procedure would happen if we stop MongoDB or even any of the nodes (for example `swarm-node-01`). If some service is not responding, Consul triggers corresponding Jenkins job and repeats the deployment cycle. Even if the whole node is down services residing on that node would not respond and the procedure would be repeated.

How does this work?



Every time we deploy a service with Ansible (at the moment only one but that could extend to any number of them) we're making sure that Consul configuration is updated with information about the service. Let's take a look at configuration for the **books-service**.

```
1 | cat /etc/consul.d/service-books-service.json
```

The output should be following.

```
1 | {
2 |   "service": {
3 |     "name": "books-service",
4 |     "tags": ["service"],
5 |     "port": 80,
6 |     "address": "10.100.199.200",
7 |     "checks": [{
8 |       "id": "api",
9 |       "name": "HTTP on port 80",
10 |      "http": "http://10.100.199.200/api/v1/books",
11 |      "interval": "10s"
12 |     }]
13 |   }
14 | }
```

It has the information about the service. Keep in mind that, unlike information stored with Consul Registrator that has the precise server and port a service is running on, this information is from the point of view of users. In other words, it points to our public IP and port (10.100.199.200) that is handled by **nginx** which, in turn, redirects requests to the server where actual service is residing. The reason for this is that in this particular case we only care that service as a whole is never interrupted. Even though we're continually deploying **blue** and **green** releases, one of them should always be running.

Finally, there is **checks** section. it tells consul to perform **http** request on

<http://10.100.199.200/api/v1/books> every **10** seconds. If server returns anything but code **200**, Consul will consider this service not working properly and initiate "rescue" procedure.

Each service should have its own service configuration file with **checks** section tailored to its specifics. There are different types of checks but, in our case, **http** is doing exactly what we need.

Next in line is **watchers.json** configuration file.

```
1 | cat /etc/consul.d/watchers.json
```

The output is following.

```
1 | {
2 |   "watches": [
3 |     {
4 |       "type": "checks",
5 |       "state": "critical",
6 |       "handler": "/data/consul/scripts/redeploy_service.sh >>/data/consul/logs/1
7 |     }
8 |   ]
9 | }
```


It tells Consul to watch all services of type **checks**. This filter is necessary since we have services registered with Consul and we want only those we specified to be checked. Second filter is **state**. We want only service in **critical** state to be retrieved. Finally, if Consul finds a service that has both **type** and **state** match, it will run the **redeploy_service.sh** command set in **handler**. Let's take a look at it.

```
1 | cat /data/consul/scripts/redeploy_service.sh
```

The output is following.

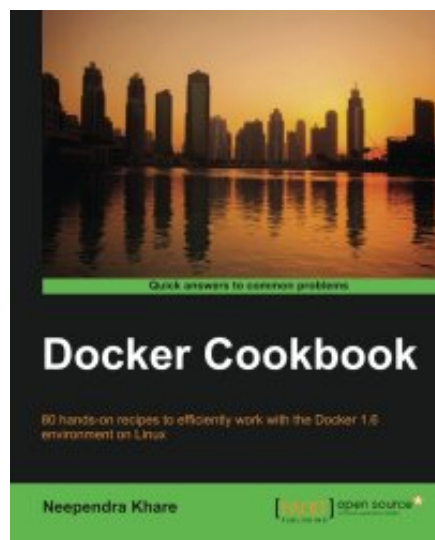
```
1 | #!/usr/bin/env bash
2 |
3 | RED='33[0;31m'
4 | NC='33[0;0m'
5 |
6 | read -r JSON
7 | echo "Consul watch request:"
8 | echo "$JSON"
9 | echo "$JSON" | jq -r '[] | select(.CheckID | contains("service:")) | .ServiceID'
10 | do
11 |     echo ""
12 |     echo -e ">>> ${RED}Service $SERVICE_NAME is critical${NC}"
13 |     echo ""
14 |     echo "Triggering Jenkins job http://10.100.199.200:8080/job/$SERVICE_NAME/bu:"
15 |     curl -X POST http://10.100.199.200:8080/job/$SERVICE_NAME/build
16 | done
```

I won't go into details of this script but say that it receives JSON from Consul, parses it to find out the name of the service that failed and, finally, makes a request to Jenkins to run the corresponding job and redeploy that service.

To Be Continued

We deployed multiple releases of our service following **blue/green deployment** technique. At no time service was interrupted during the deployment process. We have the whole process automated with Ansible and run by Jenkins.

We introduced some new Consul features that allow us to monitor the status of our services and perform measures that will recuperate them from failures. In the current setup we still cannot guarantee zero downtime since it can take up to 10 seconds for Consul to detect a failure and a bit more time for Jenkins/Ansible to perform the deployment.



The next article will go further and explore ways to scale every service into multiple nodes so that when one instance goes down, there is at least one more running. That way, even though we can redeploy any failed service, during that process a second instance is running and making sure that there is no downtime.

The story continues in the [Scaling Individual Services](#) 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-34-blue-green-deployment-automation-and-self-healing-procedure/>] by Viktor Farcic.

3 thoughts on “Scaling To Infinity with Docker Swarm, Docker Compose and Consul (Part 3/4) – Blue-Green Deployment, Automation and Self-Healing Procedure”



bunkertor

July 13, 2015 at 12:52 pm

Reblogged this on [Agile Mobile Developer](#)



Pinco Pallo (@mscavazzin)

August 7, 2015 at 2:28 pm

Hi Viktor,

for the jenkins new slave node section, I had to also to select “Launch slave agents on Unix machines via SSH” in order to be able to complete the setting.

Thank you again for this great explanation !



Viktor Farcic

Post author

August 8, 2015 at 6:10 am

Thank you for the correction.

