Kubernetes in a nutshell — tutorial for beginners

Deploy a complete application stack just in a few steps!

Katarzyna Dusza
Sep 18, 2019 · 10 min read ★

Kubernetes is gaining wide adoption. Even though a lot of us had an opportunity to work with this **container orchestrator** before, there is still a many of us, who have never played with this platform.

Currently, there is a plenty of various courses/playgrounds, which can help you start working with the Kubernetes, like official Kubernetes tutorials or katacoda. I also went through them, but in this article, you will find not only theory but also examples, which help you implement your Kubernetes resources. We will deploy a complete application stack, consisting of a database and backend, frontend parts. In the end, you will find some exercises to do. I hope you will like it .

You can look at my Kubernetes troubleshooting guide, which covers the most common beginner questions and mistakes.

For the sake of simplicity, in this article, I will use the short name for Kubernetes: k8s.

#k8s #k8s cluster #k8s objects #deploy with kubectl #scaling & rollback #exercises

• • •

Prerequisites

Before we start, we need to install some tools and make sure that our environment is ready to play with Kubernetes.

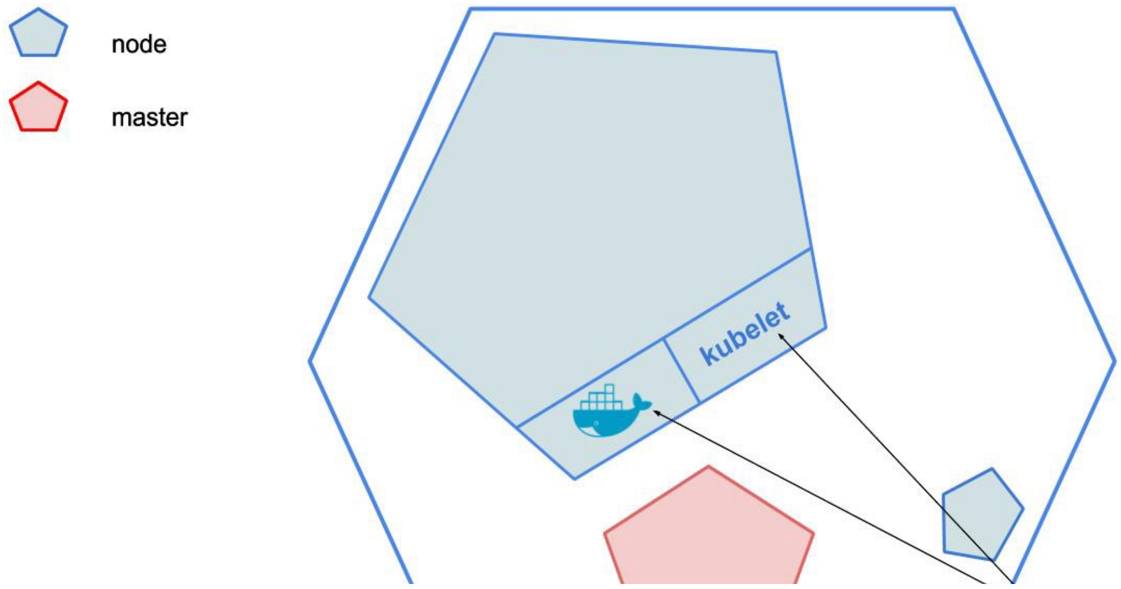
All the required tools, which have to be installed are listed in my Github repository, under the **Prerequisites** section. **Prepare your environment,** on the other hand, contains all commands, which have to be executed, before we start deploying our applications on Kubernetes.

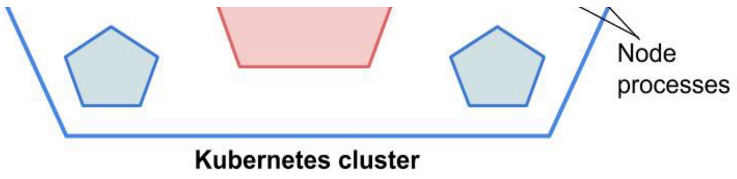
. . .

Kubernetes and its components

#Kubernetes #Master #Nodes #kubelet #node processes

Kubernetes is an open-source platform, very often called containers' orchestrator. Each k8s cluster consists of multiple components, where Master, Nodes and k8s resources (k8s objects) are the most essential ones.





Kubernetes cluster

Master is the cluster orchestrator, which exposes the k8s API (docs). Every time, when we are deploying the app containers, we are telling something like: *"Hey Master! Here is the docker image URL of my application. Please, start the app container for me."*. And then Master schedules the app instances (containers) to run on the Nodes. Kubernetes will choose where to deploy the app based on Nodes' available resources.

> *Master manages the cluster. It scales and schedules app containers and rolls out the updates.*

Nodes are k8s workers, which run app containers. A Node consists of the following processes:

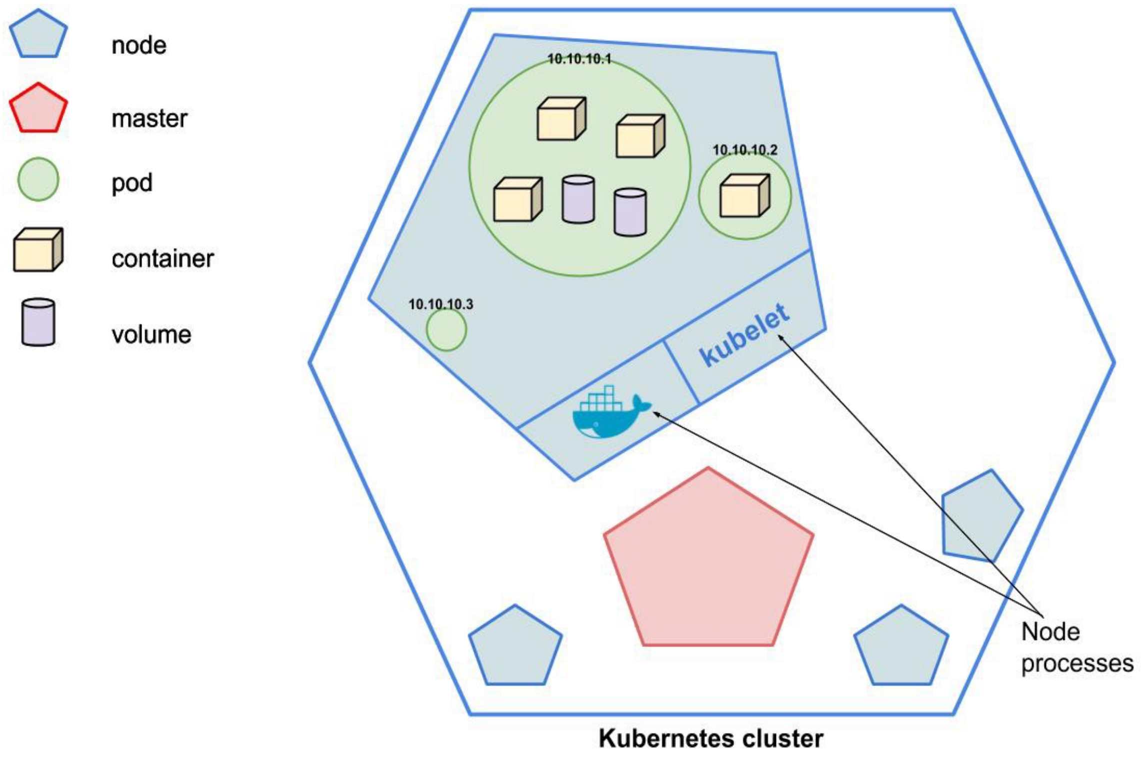
> ***kubelet*** — it's an agent for managing the Node. It communicates with the Master using k8s API. It manages the containers and ensures that they are running and are healthy.
>
> ***other tools*** — Node contains additional tools like Docker, to handle the container operations like pulling the image, running and so on.

> *Nodes are workers, which run application containers. They consist of the kubelet agent, which manages the Node and comunicates with the Master.*

. . .

Kubernetes resources

Pod



Pod in Kubernetes cluster

Pod is the smallest resource in k8s. It represents a group of one or more application containers and some shared resources (volumes). It runs on a private, isolated network, so containers can talk to each other using *localhost*. Normally, you would have one container per Pod. But sometimes, we can run multiple containers in one Pod. Typically it happens, when we want to implement side-car (read more about this pattern in Designing Distributed Systems or Dave's post).

Example:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: backend-pod-name # POD_NAME
  labels:
    application: backend # LABEL_KEY: LABEL_VALUE
spec:
  containers: # list of containers running in one Pod
    - name: main-container # CONTAINER_NAME
      image: my-backend # IMAGE_NAME
      tag: v1 # IMAGE_TAG
      ports:
      - containerPort: 8080 # CONTAINER_PORT_NUMBER - port to expose on the Pod's IP add
      env: # ENV - environment variable
      - name: NODE_ENV # ENV_NAME
        value: prod # ENV_VALUE
      imagePullPolicy: IfNotPresent # options: Always, Never, IfNotPresent;
```

simple-pod-template.yaml hosted with ❤ by GitHub view raw

To deploy a single Pod, you should run `kubectl create pod-file.yaml` or `kubectl apply -f pod-file.yaml`. Both commands are quite similar, but in comparison to `create` (which creates a resource), `apply` modifies an existing k8s object or creates new if it doesn't already exist. To see how it works, let's clone this repository and run the following command:

```
# deploy single Pod
kubectl apply -f example-pod.yaml
# see running Pod
kubectl get pods
```

Kubectl is a command-line client for k8s. It communicates with kube-apiserver (REST server), which then performs all operations.

Most of the k8s commands consist of an action and a resource, on which this action is performed:

```
# <action> represents a verb like create, delete.
# <resource> represents a k8s object.
```

```
kubectl <action> <resource> <resource-name> <flags>

# e.g.
kubectl get pods my-pod
kubectl get pods
kubectl create deployment.yaml
kubectl apply -f deployment.yaml
```

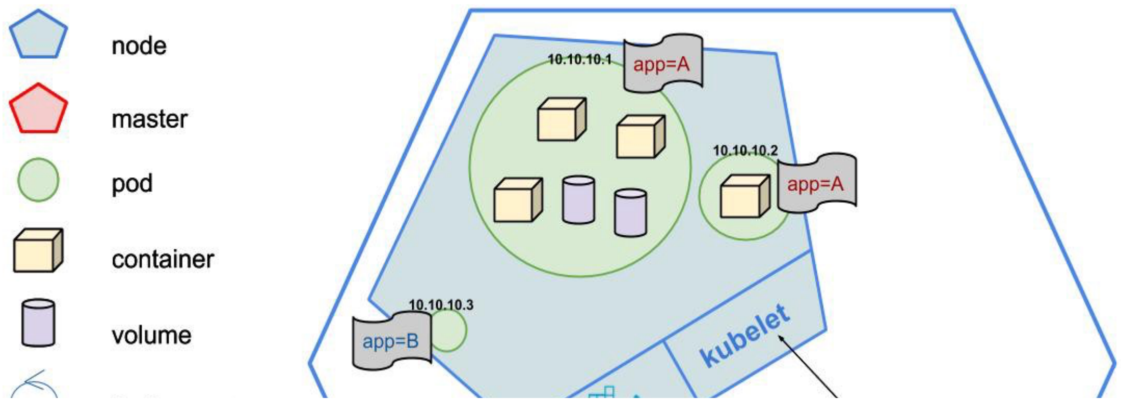
Other commands: cheatsheet and kubectl overview.

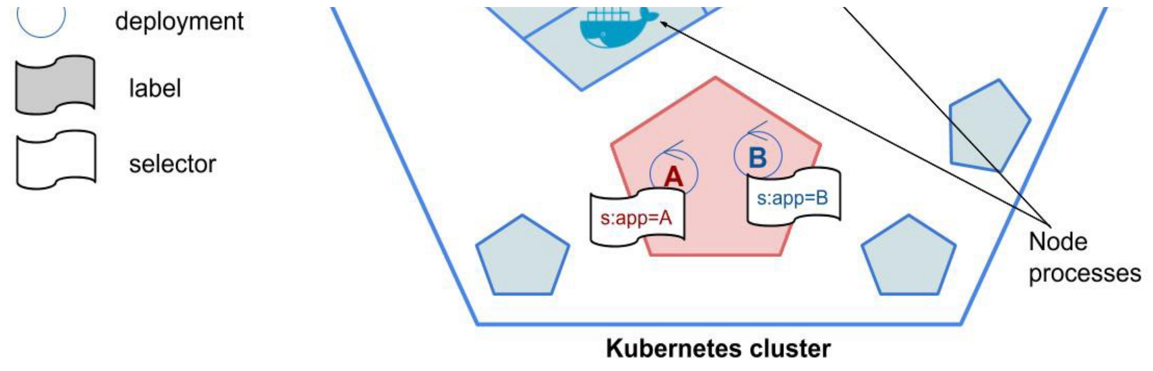
You can use short names for the k8s resources:

```
pods (or pod): po,
services (or service): svc,
deployments (or deployment): deploy,
ingresses (or ingress): ing,
namespaces (or namespace): ns,
nodes (or node): no

# example (means kubectl get pods (or kubectl get pod))

kubectl get po
```

> *Pod is the smallest resource in k8s. It runs on a private, isolated network and hosts app container instance. One Pod can contain multiple containers.*

Deployment





Deployment in Kubernetes cluster

Deployment is a k8s abstraction, which is responsible for managing (creating, updating, deleting) Pods. To deploy your application, you can always use Pods as in the previous example, however, using Deployments is a recommended way, which brings a lot of advantages:

> *you don't have to worry about managing Pods*. If one of the Pods terminates, the Deployment controller will create another Pod immediately. Deployments always take care of having a proper number of running Pods,
>
> *you have only one file, where you "define" Pod specification and desired number of running Pods*. Pod specification is under `spec.template` key, whereas number or running Pods is under `spec.replicas`,
>
> *it provides a self-healing mechanism in case of machine failure*. If the Node hosting an instance goes down or is deleted, the Deployment controller replaces the instance with an instance on another Node in the cluster,
>
> *it provides an easy way to rolling updates, etc.* If you want to apply a change to your Pods, Deployment will update all Pods gradually, one by one.

Example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
```

```yaml
  name: backend-deployment-name # DEPLOYMENT_NAME
  labels:
    # we pass selector, to easy list specific Deployments:
    # kubectl get deployment --selector=KEY_DEPLOYMENT_SELECTOR
    application: backend # KEY_DEPLOYMENT_SELECTOR: VALUE_DEPLOYMENT_SELECTOR
    resource: deployment
spec:
  replicas: 3 # NUMBER_OF_REPLICAS (number of Pods)
  selector:
    matchLabels:
      # thanks to this selector, we specify, which Pods belong to this Deployment
      application: backend # KEY_POD_SELECTOR: VALUE_POD_SELECTOR
  template: # here the Pod specification starts (look at the example-pod.yaml)
    metadata:
      labels:
        # all Pods will get this label
        # so Deployment will easily find all his Pods
        application: backend # KEY_POD_SELECTOR: VALUE_POD_SELECTOR
    spec:
      containers:
      - name: main-container # CONTAINER_NAME
        image: my-backend # IMAGE_NAME
        tag: v1 # TAG
        ports:
        - containerPort: 8080 # CONTAINER_PORT_NUMBER
```

deployment-template.yaml hosted with ❤ by GitHub view raw

Let’s create Deployments!

Before we start deploying backend and frontend, we have to make sure that the database is up and running. Our web server depends on the database’s health, so in case of issues with database connection, it will throw an error. Clone the Github repository and run the following commands to deploy the database:

```
# deploy database

kubectl apply -f database/deployment/database-deployment.yaml
kubectl apply -f database/deployment/database-service.yaml
```

Now, let's build the image and deploy backend:

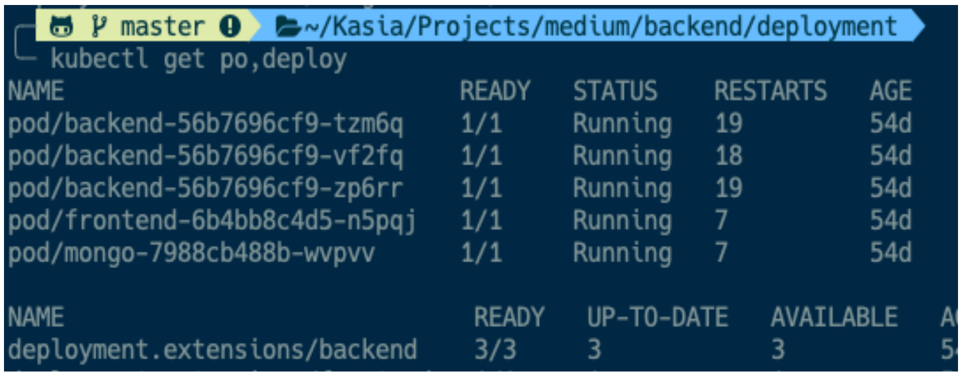
```
# build backend docker image

docker build -t backend:v1 ./backend

# deploy backend

kubectl apply -f ./backend/deployment/backend-deployment.yaml
```

Repeat the above steps to deploy the frontend. The source code with the Dockerfile is under the `frontend/` directory:

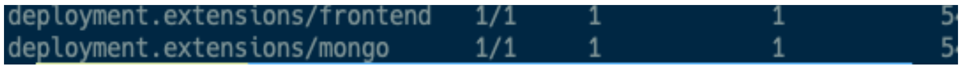
```
# build frontend docker image

docker build -t frontend:v1 ./frontend

# deploy frontend

kubectl apply -f ./frontend/deployment/frontend-deployment.yaml
```

As a result, you should see running Deployments and Pods inside the cluster:

```
kubectl get po,deploy
```



```
 master  ~/Kasia/Projects/medium/backend/deployment
└─ kubectl get po,deploy
NAME                                READY   STATUS    RESTARTS   AGE
pod/backend-56b7696cf9-tzm6q        1/1     Running   19         54d
pod/backend-56b7696cf9-vf2fq        1/1     Running   18         54d
pod/backend-56b7696cf9-zp6rr        1/1     Running   19         54d
pod/frontend-6b4bb8c4d5-n5pqj       1/1     Running   7          54d
pod/mongo-7988cb488b-wvpvv          1/1     Running   7          54d

NAME                                READY   UP-TO-DATE   AVAILABLE   A
deployment.extensions/backend       3/3     3            3           5
```

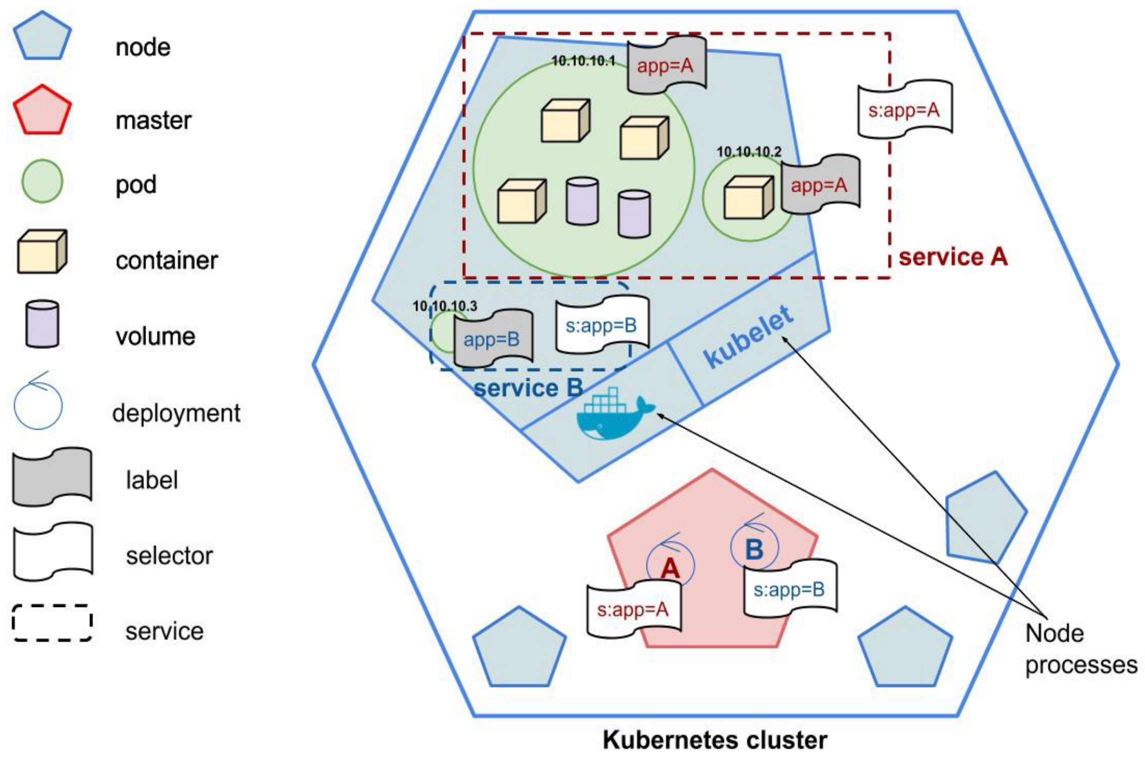


```
deployment.extensions/frontend   1/1     1            1           5
deployment.extensions/mongo      1/1     1            1           5
```

Screenshot from iTerm — kubectl get po, deploy

> *Deployment is responsible for creating and updating instances of your application. It monitors the application instance and provides a self-healing mechanism in case of machine failure.*

Service



Service in Kubernetes cluster

We already have Deployments, which started Pods with Docker containers inside. Even though they are deployed and ready to use, we can't access them. Now it's time to introduce a Service resource!

Service is another k8s object. Using it, we can make our Pods accessible from the inside, or outside k8s cluster. Service matches a set of Pods using selector defined in the Service under `spec` section and labels defined in the Pods' `metadata`. So if the Pod has the label e.g. `app: my-app`, then the Service uses this *label* as a *selector* to know, which Pods should it expose.

Currently, Service has only 4 types:

> *ClusterIP* — the default one. It allocates a cluster-internal IP address, and it makes our Pods reachable only from the inside the cluster.
>
> *NodePort* — built on top of the ClusterIP (ClusterIP on steroids). Service is now reachable not only from the inside of the cluster through the service's internal cluster IP, but also from the outside: `curl <node-ip>:<node-port>`. Each Node opens the same port (`node-port`), and redirects the traffic received on that port to the specified Service.
>
> *LoadBalancer* — build on top of the NodePort (NodePort on steroids). Service is accessible outside the cluster: `curl <service-EXTERNAL-IP>`. Traffic is now coming via LoadBalancer, which then is redirected to the Nodes on a specific port (`node-port`).
>
> *ExternalName* — this type is different than the previously mentioned. Here, you can have access to an external reference (web services/database/...) deployed somewhere else. Your Pods running in the k8s cluster can access them by using `name` specified in the Service YAML file. If you are more interested in *ExternalName* type, go to Kubernetes troubleshooting.

Example:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: backend-service-name # SERVICE_NAME
  labels:
    # we pass selector, to list specific Services:
    # kubectl get svc --selector=KEY_SERVICE_SELECTOR
    application: backend # KEY_SERVICE_SELECTOR: VALUE_SERVICE_SELECTOR
spec:
  type: ClusterIP # options: ClusterIP, NodePort, LoadBalancer, ExternalName
  selector:
    # thanks to this selector, we specify,
    # to which Pods the Service should forward the traffic
    application: backend # KEY_POD_SELECTOR: VALUE_POD_SELECTOR
  ports:
  - name: service-port # name of this port within the Service
    protocol: TCP # optional, TCP is set by default, others: UDP, SCTP
    port: 80 # SERVICE_PORT - port exposed by this service
    # Pod's port number (Pod is targeted by the Service)
    targetPort: 8080 # POD_PORT
```

simple-service-template.yaml hosted with ❤ by GitHub view raw

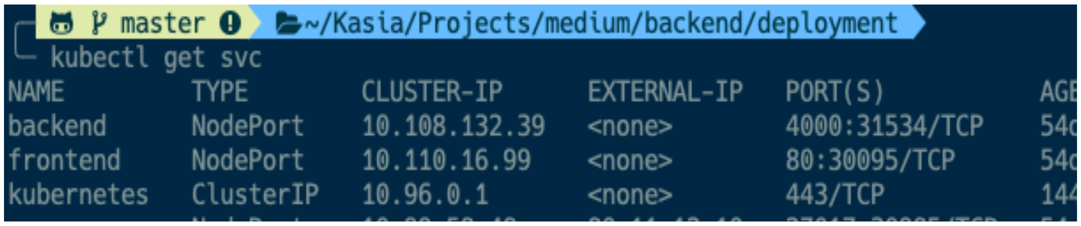
Let's create Services!

Links to the Github repository and Kubernetes troubleshooting article.

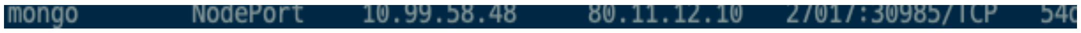
```
# deploy backend Service

kubectl apply -f ./backend/deployment/backend-service.yaml

# deploy frontend Service

kubectl apply -f ./frontend/deployment/frontend-service.yaml
```

As a result, you should see running Services inside the cluster:

```
kubectl get svc
```



```
master  ~/Kasia/Projects/medium/backend/deployment
└ kubectl get svc
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
backend      NodePort    10.108.132.39    <none>        4000:31534/TCP   54
frontend     NodePort    10.110.16.99     <none>        80:30095/TCP     54
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP          144
```

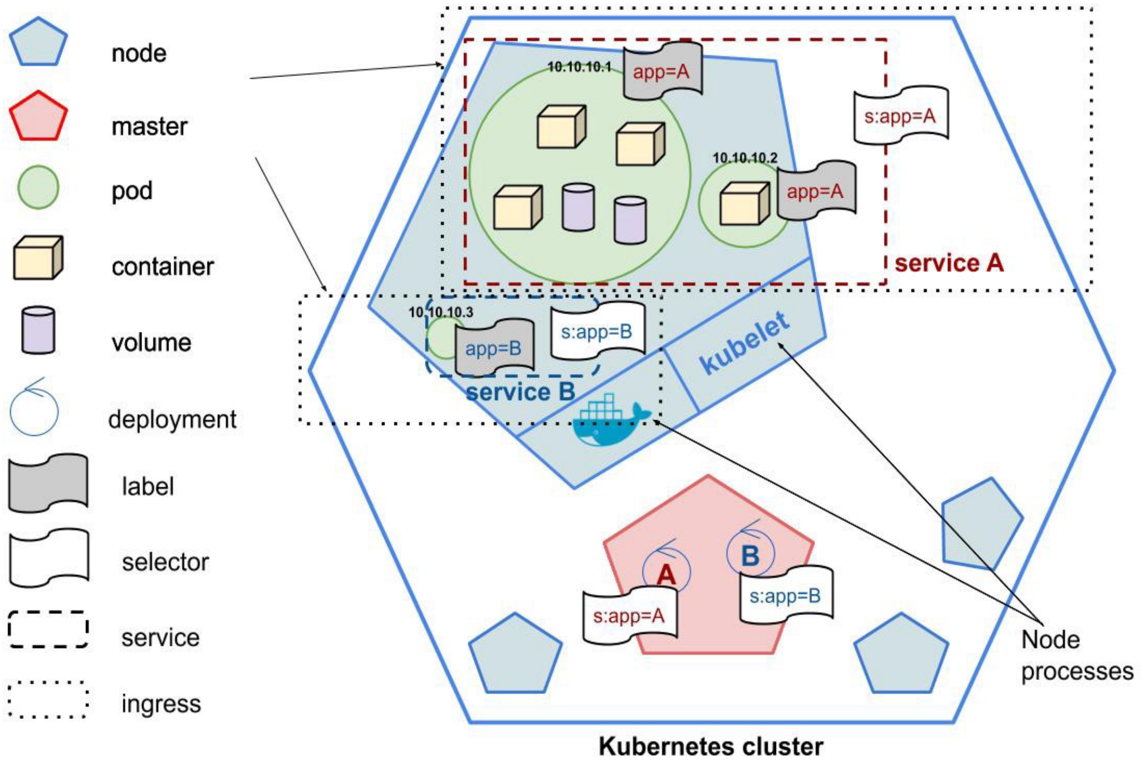


```
mongo         NodePort     10.99.58.48     80.11.12.10    27017:30985/TCP    54c
```

Screenshot from iTerm — kubectl get svc

> *Service defines the policy, by which we are able to access the Pods. Service matches a set of Pods using his selector and Pod's labels. There are 4 types of Services: ClusterIP, NodePort, LoadBalander and ExternalName. Connections to the Service are load-balanced across all the backing pods.*

Ingress



Ingress in Kubernetes cluster

Ingress is a simple proxy, which routes traffic to the Services in the cluster. In one Ingress you can specify multiple Services to which it will redirect the traffic. Potentially, we don't have to use Ingresses, but using it brings some advantages like having virtual hosts, SSL, CORS settings and so on.

Example:

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: backend-ingress-name # INGRESS_NAME
  labels:
    # we pass selector, to easy list specific Ingresses:
    # kubectl get ing --selector=KEY_INGRESS_SELECTOR
    application: backend # KEY_INGRESS_SELECTOR: VALUE_INGRESS_SELECTOR
  annotations:
    # you can apply additional annotations here, to enable cors for example
    # all example annotations (kubernetes/ingress-nginx controller) you can find here:
    # https://github.com/kubernetes/ingress-nginx/blob/master/docs/user-guide/nginx-conf
spec:
  rules:
    - host: my-app.domain.com # HOST_NAME
      http:
        paths:
          - path: / # path for the host
            backend:
              # thanks to this section, we specify,
              # to which Service the Ingress should forward the traffic
              serviceName: backend-service-name # SERVICE_NAME
              servicePort: 80 # SERVICE_PORT
```

simple-ingress-template.yaml hosted with ❤ by GitHub view raw

Let's create Ingresses!

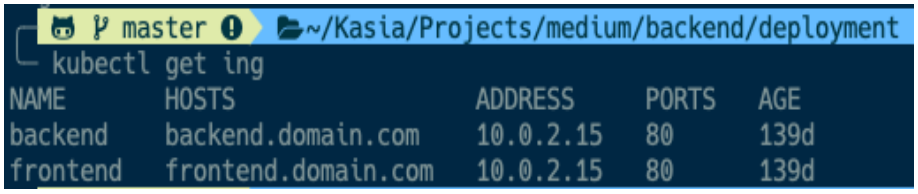
Links to the Github repository and Kubernetes troubleshooting article.

```
# deploy backend Ingress

kubectl apply -f ./backend/deployment/backend-ingress.yaml

# deploy frontend Ingress
```

```
kubectl apply -f ./frontend/deployment/frontend-ingress.yaml
```

As a result, you should see running Ingresses inside the cluster:

```
kubectl get ing
```



```
master  ~/Kasia/Projects/medium/backend/deployment
kubectl get ing
NAME       HOSTS                 ADDRESS     PORTS   AGE
backend    backend.domain.com    10.0.2.15   80      139d
frontend   frontend.domain.com   10.0.2.15   80      139d
```

Screenshot from iTerm — kubectl get ing

We have already deployed a complete application stack. Now, let's check how does it look in the browser.

> *Ingress is a simple proxy, which routes the traffic to the Services in the cluster. It is easily extensible to take care of eg. CORS settings or SSL. It's possible to have one Ingress for all Services in your cluster.*

. . .

Scaling, rollback and quick updates

Scaling

There are a couple of ways to quickly scale up or down replicas of the Deployment. You can scale them based on specific conditions, like current replicas number. If you need, you can also turn on autoscaling in the cluster. Read this resource for more details.

```
# scale the resource to specific number of replicas
```

```
kubectl scale --replicas=REPLICAS_NUMBER -f your-yaml-file.yaml

# example: scale backend-deployment.yaml

kubectl scale --replicas=3 ./backend/deployment/backend-
deployment.yaml

# scale up to 3, when the current number of replicas of deployment
is 2

kubectl scale --current-replicas=2 --replicas=3
deployment/DEPLOYMENT_NAME

# autoscale deployment between 2 - 10

kubectl autoscale deployment DEPLOYMENT_NAME --min=2 --max=10
```

More scaling commands: cheatsheet.

Rollback

Sometimes it happens that we deployed a new feature, which has a bug. If it’s something serious, engineers choose the option to roll back the current version to the previous one. Kubernetes allows us to do it

```
# rolling update "c" containers of "DEPLOYMENT_NAME" deployment,
updating the IMAGE_NAME image

kubectl set image deployment/DEPLOYMENT_NAME c=IMAGE_NAME:v2

# rollback to the previous deployment

kubectl rollout undo deployment/DEPLOYMENT_NAME

# watch rolling update status of deployment until completion

kubectl rollout status -w deployment/DEPLOYMENT_NAME
```

Quick updates

```
# update a single-container pod's image version (tag) to v4

kubectl get pod POD_NAME -o yaml | sed 's/\(image:
IMAGE_NAME\):.*$/\1:v4/' | kubectl replace -f -

# add a label

kubectl label pods POD_NAME new-label=awesome

# add an annotation

kubectl annotate pods POD_NAME icon-url=http://goo.gl/XXBTWq
```

. . .

Exercise

Now it's time for an exercise for you. Inside the `/exercise` directory, you will find a small service with Dockerfile. All you need to do is build the image and create YAMLs to deploy the app on k8s!

Good luck!

. . .

Conclusion

The tutorial, which you have just finished allows you to start working with Kubernetes. I tried to point out the most important/tricky parts, which sometimes gave me sleepless nights .

Kubernetes is a very powerful platform. When you understand the basic concepts, you can already do a lot, but you will be still hungry to learn and play more with it. Until now, I will leave you with some resources, which I encourage you to read one by one.

. . .

Resources. Nice to read

"Kubernetes in action" book, written by Marko Luksa. My favorite book about Kubernetes.

Kubernetes docs — always very helpful.

Kubernetes API reference — I very often look into the API. Sometimes it's easier to understand resources (their specification, what they have, what they need) from API than from the docs.

"Kubernetes NodePort vs LoadBalancer vs Ingress?" blog post written by Sandeep Dinesh, with a good explanation about exposing your Services, with nice pictures.