**Guide To Kubernetes – Part I**

## Overview

You will implement how to integrate Node.js applications with Docker and run them in Kubernetes clusters, we will cover the following topics:

* Dockerizing an existing Node.js application.
* Deploying it using Docker Compose.
* Integrating and deploying to a Kubernetes cluster.
* Inspecting and scaling

## Requirements

Before proceeding, make sure that your environment satisfies these requirements. Start by installing the dependencies on your machine.

* Docker
* Kubernetes
* NPM
* Git

## Run the application locally

$> docker run -d -p 27017:27017 mongo:4.2

The above command will create a MongoDB container and expose it on port 27017 of the host machine.

The next step is to clone the project repository locally and run it using the below commands:

$> git clone git@github.com:wshihadeh/node-easy-notes-app.git

$> cd node-easy-notes-app

$> npm install

$> node server.js

Once the application starts, you can check and verify that it’s running using the following curl command:

$> curl -fs http://127.0.0.1:3000

## Dockerize the application

To run our application in a Docker container, we need to create a Dockerfile that describes how to build the Docker image. The Dockerfile instructions should cover the following points

* The base Docker image.
* Coping the application source code to the Docker image.
* Installing the application’s dependencies.
* Setting up the application start command.

The below Dockerfile can be used to build the Docker image for the Node.js application. The file needs to be stored in the root directory of the application.

FROM node:12.0-slim

COPY . .

RUN npm install

CMD [ “node”, “server.js” ]

Building the Docker image for the application can be done using the below command

$> docker build -t ${namespace}/{imagename}:{tag}

It’s a common practice to manage the build of Docker images using a makefile. The main benefit behind this recommendation is that a makefile helps in providing a standard interface to build Docker images and hide the complexity of its commands.

This task can be achieved by adding the following Makefile to the repository.

# Docker registry

REGISTRY ?= index.docker.io

#Image namespace

NAMESPACE ?= default\_name\_space

# image name

NAME ?= node-easy-notes-app

#image default tag

IMAGE\_TAG ?= latest

IMAGE\_NAME = ${REGISTRY}/${NAMESPACE}/${NAME}:${IMAGE\_TAG}

build:

docker build -t ${IMAGE\_NAME} .

Building the Docker image now can be done simply using one of the following commands based on the needs

$> make build # this command will use the default values

$> IMAGE\_TAG=v1 make build # this command will use v1 as an image tag

## Run the application using Docker Compose

Docker Compose is a tool for defining and running multi-container Docker applications.

With Compose, the application’s services are defined and configured in a YAML file. Then, with a single command, services can be created and started.

To be able to run the Node.js applications with docker-compose we need to define two services within the docker-compose file: one for the MongoDB service and the second for the Node.js application. The below compose file can be used to deploy both services.

version: '3.7'

volumes:

* mongo-db: :/data/db

networks:

nodeJsNet:

external: false

name: 'nodeJsNet'

services:

mongodb:

image: mongo:4.2

restart: always

networks:

- nodeJsNet

volumes:

- mongo-db:/data/db

easy-notes:

image: node-easy-notes-app:latest

restart: always

networks:

- nodeJsNet

environment:

MONGO\_URL: 'mongodb://mongodb:27017/easy-notes'

ports:

- 8080:3000

As is shown in the above snippet, the docker-compose have the following configurations

* Two services are defined, one for MongoDB and the other for the Node.js application.
* Both applications are connected to the same network, and as a result, they can reach each other.
* MongoDB is configured to use a volume to store the data. As a result, when the container crashes or exits, the data will not be lost.
* The Node.js application is configured to use the MongoDB instance via environment variables.
* The Node.js application is accessible from the host machine on port 8080.

It is easy to learn and deploy Docker services with docker-compose due to the fact the all the service definitions and configurations can be included in a simple single YAML file. In addition, docker-compose helps in creating and working with applications for development environments.

However, it is not recommended to use docker-compose for deploying, maintaining and managing production Docker services because it is missing a lot of features needed for production services such as support for zero-downtime deployment and running services in Docker clusters natively. On the other hand, container orchestrators such as Docker swarm and Kubernetes are designed to solve and handle Docker production workloads. Below are some advantages of using container orchestrators:

* They are built to run complex applications with a large number of microservices.
* They have native support for many deployment-related features such as zero-downtime deployments and resource management.
* They support clustering Docker nodes, high availability, and auto-recovery.

In the next part, we will start using Kubernetes, one of the most widely known container orchestration frameworks. Then we will dive deep in the details of deploying and scaling Kubernetes Services.

## Kubernetes Overview

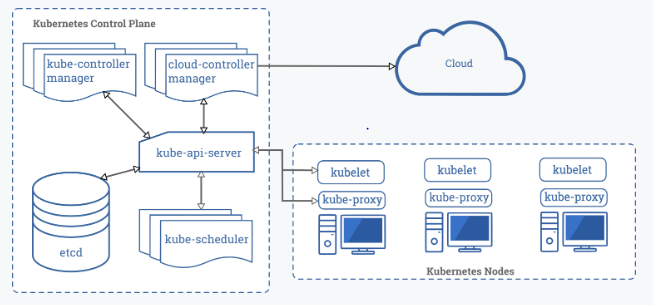
Kubernetes (also called K8s) is an open-source system for automating deployment, scaling, and management of containerized applications. Kubernetes is commonly used by many individuals and companies for the following reasons:

* **Open Source**: The project is totally free. You can download, install and use it freely. Moreover, you can access and contribute to its source code.
* **Huge community and support**: Many individuals and well-known companies such as Redhat, Google, Microsoft, IBM, and Cisco contribute to K8s source code. This adds a layer of confidence to the quality of the source code.
* **Managed cloud solutions**: Many cloud providers such as AWS, Google and DigitalOcean are providing Kubernetes managed services.
* **A multitude of features**: Kubernetes supports a wide range of features such as isolating applications in namespaces, network policies, storage options, rolling updates, and many other capabilities.

Kubernetes consists of multiple components and each of these components are designed to serve a specific purpose. Below is a brief description of Kubernetes components

* **Etcd cluster**: stores information about the cluster
* **Kube scheduler**: responsible for scheduling applications or containers on nodes
* **Controllers**: take care of different functions like the node control and replication controller.
* **Kube API server**: responsible for orchestrating all operations within the cluster.
* **Kubele**t: the primary node agent. It listens to the instructions from the Kube-api-server and manages containers in the registered nodes.
* **Kube-proxy**: helps in enabling communication between services within the cluster.
* **Kubectl**: the acronym of Kubernetes command-line client and it’s used to create, edit, update, delete, and view Kubernetes resources.

More information about the Kubernetes component and their roles can be found on the official [Kubernetes website](https://kubernetes.io/docs/concepts/overview/components/). Below is a diagram that shows a Kubernetes cluster with all the components tied together.



## Kubernetes Resources

Using the Kubernetes command-line tool kubectl (also using other k8s clients or directly using the kube-api-server), API resources can be created on Kubernetes.

Kubernetes supports more than 50 different resource types for managing the cluster such as Deployment and Service. You can view the full list using kubectl:

$> kubectl api-resources

It is not mandatory to use all the defined resources to deploy services in a Kubernetes cluster. In fact, the usage of Kubernetes resources is highly dependent on the nature and requirements of the services. Below is a brief summary of the most used resources and the ones that we will be using during this post.

* A Pod is the basic execution unit of a Kubernetes application. Pods can run one or multiple Docker containers.
* A ReplicaSet is responsible for setting and managing replications of Pods and maintaining a stable state of the replications.
* A Deployment provides us with the capability to upgrade the underlying instances seamlessly using rolling updates, undo changes, and pause and resume changes as required.
* A Service enables the communication between various components within and outside of the application.
* The Ingress manages external access to the services in a cluster
* A PersistentVolume is a piece of storage in the cluster that has been provisioned.
* A PersistentVolumeClaim is a request for PersistentVolume by a user or an application.

## Combining the powerful features of Node.js, Docker, microservices, and Kubernetes

If you are using Node.js to develop your applications, you’re very likely building microservices. Node.js is a lightweight technology that goes well with microservices architecture. Many built-in Node.js features allow communicating with other services such as databases in a performant and fast way.

To obtain the full advantages of microservices architecture, using Docker and a robust orchestration technology like Kubernetes, is certainly the best choice.

----------------------------------- Part II ----------------------

## Deploying MongoDB

It is possible to deploy applications in Kubernetes using a single imperative command as shown below. However, it is recommended to use the declarative way of deploying applications and use deployment objects instead of Pods directly for two main reasons:

* The declarative method is easier to review, automate, and backup
* Deployment objects automate the replication of the Pods and rolling out updates of the application

$> kubectl run --generator=run-pod/v1 --image=mongo:4.2 mongo-db

To deploy MongoDB to Kubernetes we need the following resources:

### **Persistent Volume**

This resource is needed to define the storage volume where MongoDB data can be stored. Kubernetes supports a wide range of PersistentVolumes such as Glusterfs, CephFS, AzureFile, and [many more](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#types-of-persistent-volumes). For the sake of simplicity, we will use the HostPath [volume plugin](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#types-of-persistent-volumes) to create the volume locally.

Usually, we use YAML files to define resources and the kubectl command-line to create, manage and update the same resources. There are four common configurations sections among Kubernetes resources.

These sections are listed below

* **apiVersion**: describes the Kubernetes API versions to be used to create the resource.
* **Kind**: describes the type of resource to be created.
* **Metadata**: attaches meta-information regarding the resource such as its name or labels.
* **Spec**: describes the specifications of the resource(s).

We are going to use the following YAML to create a PersistentVolume object in Kubernetes. As it’s shown, the definitions file follows the same structure described above. In the Spec section, the object details and configurations are provided such as the capacity and the path of the volume on the host machine.

apiVersion: v1

kind: PersistentVolume

metadata:

name: mo-data-pv

labels:

type: local

spec:

storageClassName: generic

capacity:

storage: 500Mi

accessModes:

- ReadWriteOnce

hostPath:

path: "/var/lib/mongo"

Creating the above object can be done by executing one of the following commands

$> kubectl apply -f mongo-pv.yaml

$> kubectl create -f mongo-pv.yaml

And you can verify the creation of the resource and its status using the following commands:

$> kubectl get persistentvolumes mo-data-pv

$> kubectl describe persistentvolumes mo-data-pv

Note that we used the resource name defined in the resource file.

### **Persistent Volume Claim**

Each application that needs to store data in a volume needs to request access to a data volume. This action can be achieved by creating and attaching a PersistentVolumeClaim to the application Pods. The below snippet shows the definition file that can be used to create the PersistentVolumeClaim. The same sections are included in the file but with different values and configurations

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: mo-data-pvc

spec:

storageClassName: generic

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 500Mi

Creating the above object can be done by executing one of the following commands:

$> kubectl apply -f mongo-pvc.yaml

$> kubectl create -f mongo-pvc.yaml

And you can verify the creation of the resource and its status using the following commands:

$> kubectl get service mongodb-service

$> kubectl describe service mongodb-service

$> kubectl get endpoints mongodb-service

Now that we are done with deploying the MongoDB resources, let’s deploy the Node.js application.

## Deploy the Node.js App

The deployment of this application requires a Deployment and a Service object since it’s stateless and does not persist data (in contrast to the MongoDB application where data is persistent).

### **Deployment**

The deployment resource for our Node.js application is similar to MongoDB one with minor differences such as labels and Deployment name. In addition, the Node.js container spec defines different environment variables. Note that we use the MongoDB service name “mongodb-service” to configure the connection between Node.js and MongoDB.

apiVersion: apps/v1

kind: Deployment

metadata:

labels:

app: easy-notes

name: easy-notes

spec:

replicas: 1

selector:

matchLabels:

app: easy-notes-pod

template:

metadata:

labels:

app: easy-notes-pod

spec:

containers:

- name: easy-notes

env:

- name: MONGO\_URL

value: mongodb://mongodb-service:27017/easy-notes

image: wshihadeh/node-easy-notes-app:latest

ports:

- containerPort: 3000

restartPolicy: Always

### **Service**

The Node.js service can be created using the below definition file which exposes the application service on port 8080 and redirects the traffic to the containers on port 3000.

apiVersion: v1

kind: Service

metadata:

labels:

app: easy-notes

name: easy-notes-service

spec:

ports:

- port: 8080

targetPort: 3000

selector:

app: easy-notes-pod

type: ClusterIP

Creating and managing the Node.js deployment and service should be done in a similar way to what we did with MongoDB.

### **Expose the service externally**

Both the MongoDB and the Node.js applications are running on Kubernetes, however, they are only accessible from inside the cluster. In order to access Node.js form host nodes or from the outside, we need to implement one of these options

* Update the Node.js service to be a NodePort service instead of ClusterIP.
* Use an Nginx Ingress controller to expose the service.

Using the Nginx controller is the best option. It helps in reducing the number of exposed and managed ports and allows controlling the incoming traffic to our services. Therefore, we will implement this option. Below are the steps needed for deploying and using Nginx ingress controllers.

First of all, we need to deploy a default backend application. This can be any service that responds with a 404 page at “/” and 200 on a “/healthz” endpoints. Below is an example of the resources needed to deploy such an application:

apiVersion: apps/v1

kind: Deployment

metadata:

name: default-http-backend

labels:

app: default-http-backend

spec:

replicas: 1

selector:

matchLabels:

app: default-http-backend

template:

metadata:

labels:

app: default-http-backend

spec:

terminationGracePeriodSeconds: 60

containers:

- name: default-http-backend

image: gcr.io/google\_containers/defaultbackend:1.4

livenessProbe:

httpGet:

path: /healthz

port: 8080

scheme: HTTP

initialDelaySeconds: 30

timeoutSeconds: 5

ports:

- containerPort: 8080

---

apiVersion: v1

kind: Service

metadata:

name: default-http-backend

labels:

app: default-http-backend

spec:

type: ClusterIP

ports:

- port: 80

targetPort: 8080

selector:

app: default-http-backend

The next step is deploying an Nginx Ingress Controller. The Controller can be deployed using a Deployment resource. It is important to configure it with a default backend and set the POD\_NAME and POD\_NAMESPACE environment variables.

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-ingress-controller

namespace: default

spec:

replicas: 1

selector:

matchLabels:

app: ingress-nginx

template:

metadata:

labels:

app: ingress-nginx

spec:

containers:

- name: nginx-ingress-controller

image: quay.io/kubernetes-ingress-controller/nginx-ingress-controller:0.13.0

args:

- /nginx-ingress-controller

- --default-backend-service=default/default-http-backend

- --annotations-prefix=nginx.ingress.kubernetes.io

env:

- name: POD\_NAME

valueFrom:

fieldRef:

fieldPath: metadata.name

- name: POD\_NAMESPACE

valueFrom:

fieldRef:

fieldPath: metadata.namespace

ports:

- name: http

containerPort: 80

- name: https

containerPort: 443

Now, let’s expose the Nginx Controller on port 80 and 443. To do this, define a Service resource for the Nginx Controller. We need to expose a LoadBalancer on the host nodes (you can also use NodePort).

apiVersion: v1

kind: Service

metadata:

name: ingress-nginx

spec:

type: LoadBalancer

ports:

- name: http

port: 80

targetPort: 80

protocol: TCP

- name: https

port: 443

targetPort: 443

protocol: TCP

selector:

app: ingress-nginx

Lastly, configure the Ingress Controller to expose and forward traffic to the Node.js application.

This task can be achieved by creating an Ingress resource that defines how and when to forward the traffic to the Node.js application. With the below resource we configured the Ingress Controller to forward traffic to the application Service on port 8080 when the host of the request is easynotes.lvh.me.

apiVersion: networking.k8s.io/v1beta1

kind: Ingress

metadata:

name: easy-notes-ingress

spec:

rules:

- host: easynotes.lvh.me

http:

paths:

- backend:

serviceName: easy-notes-service

servicePort: 8080

Now, create the defined resources:

$> kubectl apply -f ${file\_name}

$> kubectl create -f ${file\_name}

Once all Services are deployed you should be able to access our Node.js application form the host machine using your web browser or the following curl command:

$> curl -fs easynotes.lvh.me

## Scale the services

In the previous sections, we created a Deployment, and then exposed it publicly via a Service. However, the Deployment created one Pod. Pod like any other physical or virtual resource has its performance limit. Imagine what happens when the workload or the external traffic increases: The Pod will certainly go down and our application Service will fail to serve the users. This is when scaling can help. In this kind of scenario, increasing the number of Pods is a feature that allows you to handle more traffic. This is why, as a developer, you should learn about scaling your Services.

Fortunately, this powerful feature is achieved easily. Scaling any of the services described in this post can be done using the below command.

$> kubectl scale deployment --replicas ${replica\_count} ${deployment\_name}