NodeJS app with Kubernetes

Łukasz Wolnik Follow
May 12 · 6 min read ★

This is a simple tutorial on how to **push** a NodeJS app — with a MySQL database — **live** on https://app.your-domain.com with a valid SSL certificate under Kubernetes cluster. No prior knowledge on Kubernetes is required to follow it.

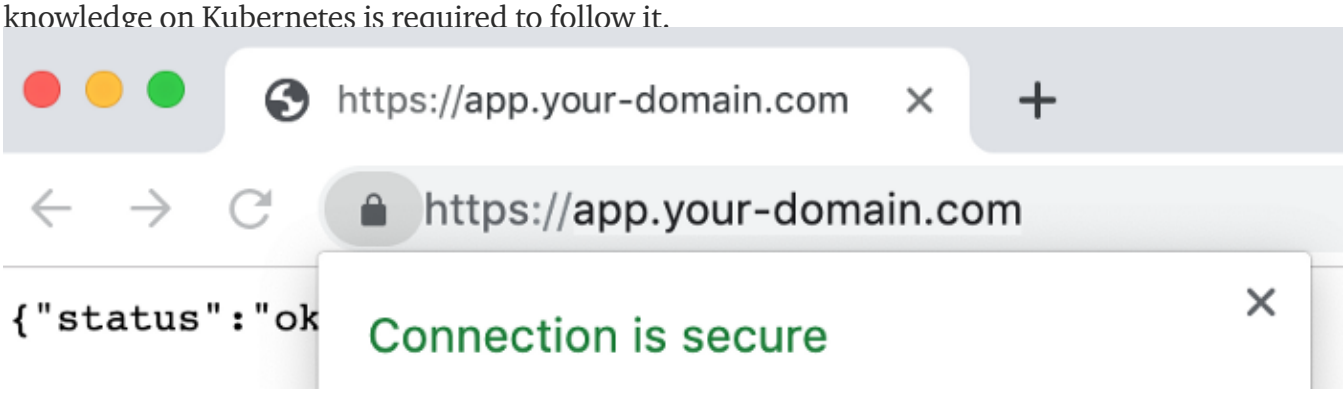

Fig 1. Your app running on a Kubernetes cluster

My minimal requirements for an app to be ready for production are:

- it's must be **private**, i.e. pulled from a private Docker repository
- it's **accessible over HTTPS** (port 443) with a valid SSL certificate
- it has an **access to** a persistent **database** (in this case, MySQL)

What I won't cover here is advanced load balancing, CDN, analytics, monitoring, backups, DDoS mitigation, etc.

In this tutorial you will learn how to:

- package your app into a Docker image
- push the image to a private Docker repository
- setup a database on a persistent volume
- run your app on Kubernetes and connect it with a database

Or visually speaking we will transition from a well-known setup depicted below:

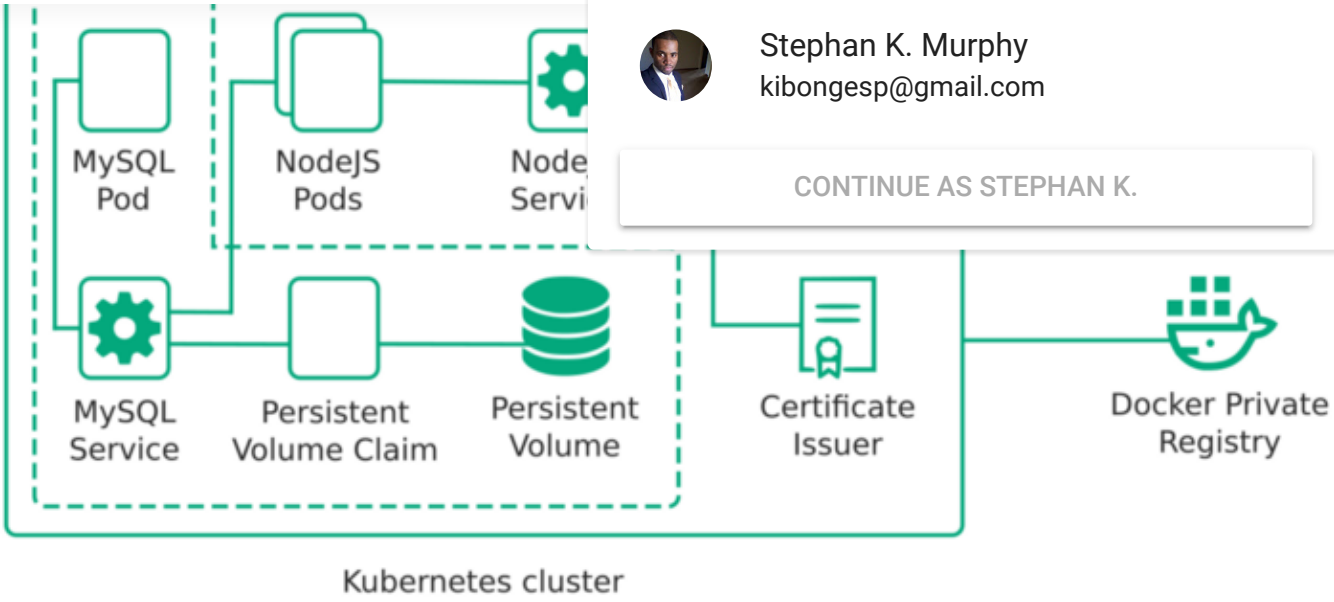

Fig 3. Result of automating installation and configuration of a typical MySQL, NodeJS app in Kubernetes cluster.

Prerequisites

The only requirement for this tutorial is to have a Kubernetes cluster running, i.e. it must have at least one worker node.

If you haven't got an access to Kubernetes you can either create your own cluster or if you don't want to master the `kubeadm` command just yet use a paid provider (Amazon, Azure, OVH) that will deal with all the setup and maintenance of one's cluster.

Kubernetes cluster

You can easily check if you have an access to a Kubernetes cluster using `kubectl` command in your terminal. Make sure you have `kubectl` installed on your system first.

```
$ kubectl cluster-info
```

You should see an output beginning with:

```
Kubernetes master is running at https://k8s.example.com
KubeDNS is running at
https://k8s.example.com/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
```

Dockerize your app

Kubernetes requires an app to be packaged into a container. By default it's using Docker containers.

If your project is not an open-source one yo
private repository. Docker Hub offers one p
more private repos you may want to use a f
repos.
On your local machine sign in to your Dock

```
docker login cloud.canister.io:5000
```

It will save your auth token in your home directory, i.e. `~/.docker/config.json`.
Now build your NodeJS app's Docker image and push it to your private repository.
Your Kubernetes cluster also needs to be authorised to pull your NodeJS app's private Docker image. The way to store sensitive data on Kubernetes are configuration YAML files named Secrets. To generate a Secret — named `regcred` — on our cluster that contains our Docker private repo's authentication token run:

```
kubectl create secret generic regcred --from-
file=.dockerconfigjson=~/.docker/config.json --
type=kubernetes.io/dockerconfigjson
```

Now every time your deployment will point to an Docker image on a private repo, Kubernetes will use the auth token stored in `regcred` to obtain access to it.

Deploy your NodeJS app

To deploy your app on your Kubernetes cluster create a deployment instructions in a `node.yml` file that reads:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: node-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: node
  template:
    metadata:
      labels:
        app: node
    spec:
      containers:
```

```
  ports:
    - port: 80 # expose the se
      targetPort: 3000 # our N
```

Deploy on your Kubernetes cluster:

```
kubectl apply -f node.yml
```

MySQL database

Deploying a database on a cluster doesn't differ from deploying your NodeJS app. The only difference is that we want our database to persist the data whenever the MySQL pod is destroyed and recreated. And for that we need a persistent storage where database's data will be stored on a permanent physical disk.

Kubernetes uses Persitent Volume (PV) and Persistent Volume Claim (PVC) to describe a persistent storage. PVC is what deployments refers to when they need a persistent volume.

Below is an example of 10GB persistent volume (refer to your Kubernetes provider for a proper `storageClassName)`. For OVH provider it's `cinder-high-speed`, i.e. an SSD. Create a new file `mysql-pv.yml` and copy & paste below:

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: mysql-pv-volume
  labels:
    type: local
spec:
  storageClassName: cinder-high-speed # CHANGE HERE
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/data"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
spec:
  storageClassName: cinder-high-speed # CHANGE HERE
  accessModes:
```

```
  name: mysql
spec:
  type: NodePort
  ports:
    - port: 3306
    targetPort: 3306
    nodePort: 31306 # exposed
  selector:
    app: mysql
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
        - image: mysql:5.6
          name: mysql
          env:
            - name: MYSQL_ROOT_PASSWORD
              value: yourR4ndomP455w0rd
          ports:
            - containerPort: 3306
              name: mysql
          volumeMounts:
            - name: mysql-persistent-storage
              mountPath: /var/lib/mysql
      volumes:
        - name: mysql-persistent-storage
          persistentVolumeClaim:
            claimName: mysql-pv-claim
```

On the last line we refer to the Persistent Volume Claim prescribed in the `mysql-pv.yml` file.

```
  name: node-ssl-prod
spec:
  acme:
    server: https://acme-v02.a
    email: your@email.com # CH
    privateKeySecretRef:
      name: node-ssl-prod
    http01: {}
```

Run the issuer on your cluster:

```
kubectl create -f ./prod-issuer.yml
```

Grande finale

Expose your NodeJS app using Ingress, which is a routing mechanism for HTTP and HTTPS requests hitting your Kubernetes cluster.

Install `ingress-nginx` on your cluster using Helm:

```
helm install stable/nginx-ingress --namespace kube-system --name
nginx-ingress
```

Create `ingress-node.yml` file with below content:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: node-ingress
  annotations:
    kubernetes.io/ingress.class: nginx
    certmanager.k8s.io/cluster-issuer: node-ssl-prod
spec:
  tls:
    - hosts:
        - app.yourdomain.com
      secretName: node-ssl-prod
  rules:
    - host: app.yourdomain.com
      http:
        paths:
          - path: /*
            backend:
              serviceName: lockjs
```

a browser.

Conclusion

Comparing schematics of the infrastructure
moving to Kubernetes was worth it for such
I believe it was even for a personal project li
apps on Kubernetes is its requirement to des
tutorial we used the following files: `node.yml`, `mysql-pv.yml`, `mysql.yml`, `prod-issuer.yml` and `ingress-node.yml`.

Quite a lot for a little NodeJS app but it allowed us to encapsulate all the infrastructure requirements in a standardised way using Kubernetes YAML files. Which in turn liberates us to use any cloud provider (including our very own) without being forced to be locked-in in AWS, Azure or Google vendor's specific solutions. You don't need to spend any extra time to learn new cloud provider's tools of trade. You can simply feed a new Kubernetes cluster with a handful of files and Kubernetes will do the rest.

Kubernetes | Nodejs | Docker | Tutorial | DevOps

WRITTEN BY

Łukasz Wolnik

React / JavaScript / IoT developer