Member-only story

Hosting Outline VPN Server on Kubernetes

Lexa · Follow
Published in ASL19 Developers
10 min read · May 17, 2019

Listen Share More

In this document, we will look at the steps we need to take in order to host the Outline Server application on Kubernetes. The current method in use requires manual configurations for scaling and does not guarantee efficient use of the available resources. The objective here is to automate scaling or parts of it and increase efficiency.

> *Outline makes it easy for news organizations to set up a virtual private network (VPN) on their own server. — Outline*

Benchmark

Looking at Jigsaw's Outline Server repository on GitHub, we can see that the anticipated method of hosting this service is performing a "run action" on the application on a remote server — Digital Ocean or other. This will build the repo, attach the required resources and certificates, and then run the application. For doing so, there are two methods available:

1. Build and run as a Node.js app.
2. Build the application as a Docker image and run the application's container.

Motivation

We are aiming to provide this service in High Availability (HA) with the option to introduce redundancy and to easily scale up or down. Our solution consists of container orchestration and autoscaling groups. We will be using Google Kubernetes

Engine (GKE), while this pipeline may be deployed onto other Kubernetes providers (e.g. Amazon EKS, Digital Ocean, etc).

Approach

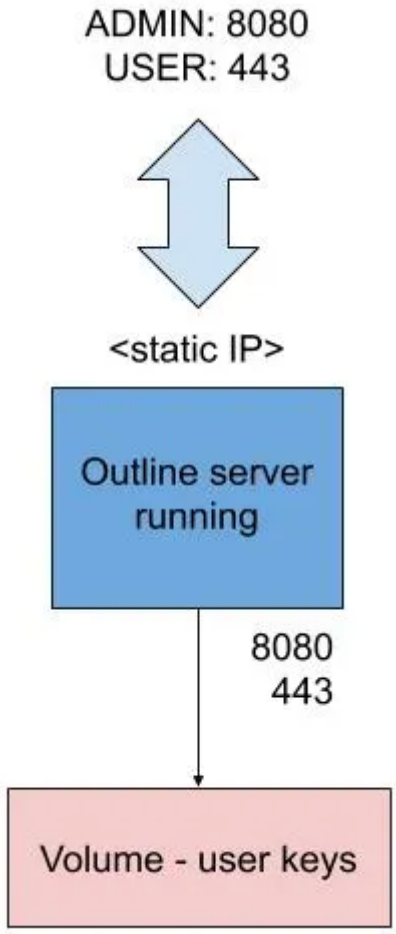
In our solution, we will employ a similar pipeline to the "ru[illegible]" [illegible] the Docker approach. Let's dissect this process taken from t[illegible] repository. There are 3 main steps to this pipeline:

1. **Build a Docker image using the "build action" script**
2. **Create a TLS certificate using OpenSSL**
3. **Run Docker image with associated environment variables and volume mounts**
4. **Output associated encrypted string to use in Outline Manager**

The steps we will be taking follow after this pipeline with the twist of breaking this service into 3 major parts, user-facing LoadBalancer, pods that host the server, and a manager pod that is only to be used for generating new VPN keys (diagram shown later in this doc).

1. **Build a Docker image using the "build action" script, tag image and push to an image registry**
2. **Create a TLS certificate using OpenSSL and copy to Kubernetes**
3. **Build Kubernetes application for VPN key generation (run Docker image with associated environment variables and volume mounts)**
4. **Build Kubernetes application for hosting the VPN server (run Docker image with associated environment variables and volume mounts)**
5. **Output associated encrypted string to use in Outline Manager**

In order to understand why we have broken running the Docker image in two steps, let's take a look at the current architecture for running Outline server versus one we are employing.



Outline Remote Server Architecture

Current Structure

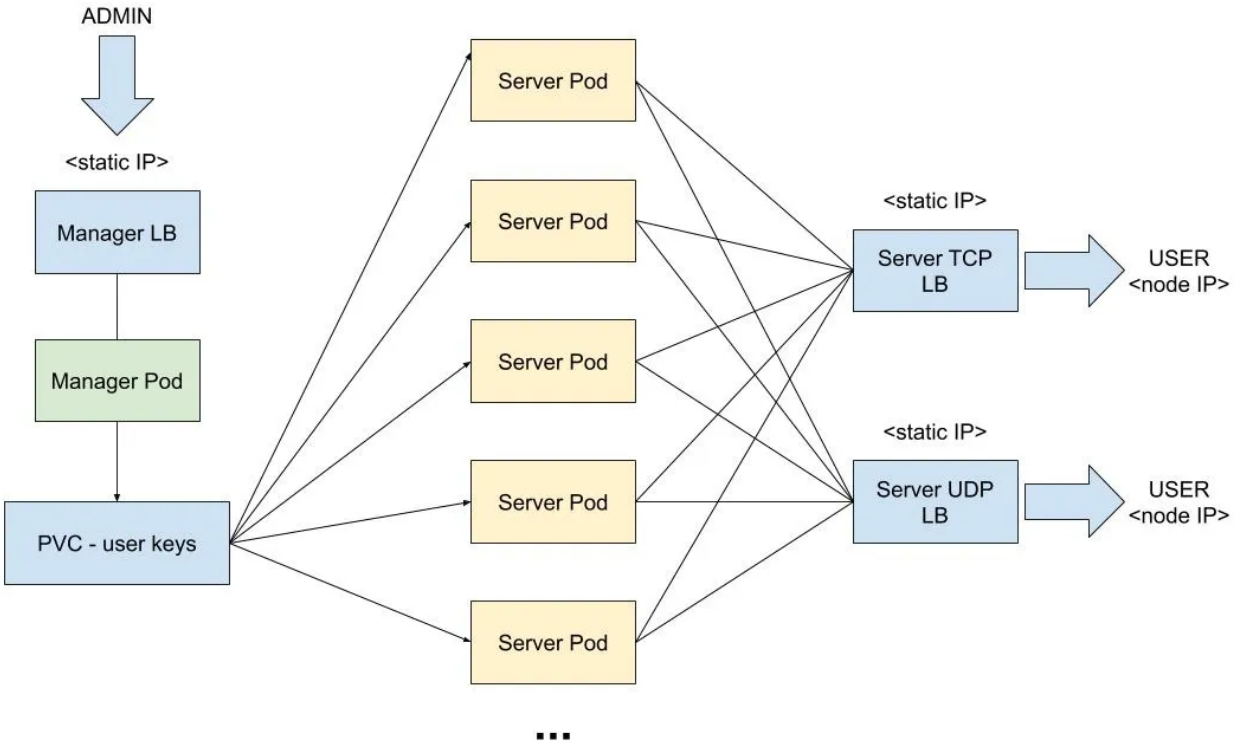
In this structure, a remote server is used, on which we run the server application. This instance of the application acts as a key generator and VPN server. The IP address that is used by the admin in the Outline Manager to generate keys, is the same as the IP address the user will use in their keys and the one the user's computer will adopt once connected. This IP address is the remote server's public IP.

Kubernetes Structure

The structure we have designed for our Kubernetes deployment of the Outline server separates the application instance that the admin uses for generating keys (**Manager pod**), from the one(s) used as the VPN server (**Server pods**). Both types of pods run the same docker image of Outline Shadowbox. This way only users with access to the Manager LB's IP address will be able to create and manage keys. In order to increase safety and cut public access to the Manager pod, we can replace the Manager LoadBalancer service with a ClusterIP and use port-forwarding to generate new keys.

The generated keys are required to be shared with the Server pods. This is accomplished through mounting the key volume onto all of the Server pods and linking the key files (config.yml and shadowbox_server_config.json) to those of the Manager pod's.

The server pods all serve the same purpose, which is to pro VPN service. These pods have their port 443 open only so that they cannot be used to generate keys since the Manager API port is set to 8080 in the environment variables. In order to allow multi-protocol incoming traffic, we then create two LoadBalancers (UDP, and TCP). They both map to the same public IP address.



Outline Kubernetes architecture

1. Build a Docker image using the "build action" script, tag image and push to a private registry

This step will help build the Outline Server Docker image with all the required settings. Clone the repository on your local computer, edit the Docker content trust variable to 0 and run build action:

```
$ git clone https://github.com/Jigsaw-Code/outline-server.git
$ cd outline-server
$ vim src/shadowbox/docker/build_action.sh
### CHANGE 1 to 0 in the export command:
```

```
### export DOCKER_CONTENT_TRUST=${DOCKER_CONTENT_TRUST:-0}
$ yarn do shadowbox/docker/build
```

You can verify that the image is built, run the following and look for the image `outline/shadowbox:latest` :

```
$ docker images
```

Tag your image to conform to your image registry naming format:

```
$ docker tag outline/shadowbox:latest <registry>/<project path>:
<version>
$ docker login <registry>
$ docker push <registry>/<project path>:<version>
```

You have now successfully pushed the Outline server Docker image to your private image registry.

2. Create a TLS certificate using OpenSSL and copy to Kubernetes

This step assumes you have successfully created a GKE cluster and added the context to your local `kubeconfig` file, as well installed `kubectl` and `gcloud` CLI tools. You can follow this document in case you do not already have a cluster ready. Once done, run the following:

```
export CERTIFICATE_NAME=shadowbox-selfsigned-dev
export SB_CERTIFICATE_FILE="cert/${CERTIFICATE_NAME}.crt"
export SB_PRIVATE_KEY_FILE="cert/${CERTIFICATE_NAME}.key"
mkdir cert

declare -a openssl_req_flags=(
  -x509
  -nodes
  -days 36500
  -newkey rsa:2048
  -subj '/CN=localhost'
  -keyout "${SB_PRIVATE_KEY_FILE}"
  -out "${SB_CERTIFICATE_FILE}"
)
openssl req "${openssl_req_flags[@]}"
```

```
kubectl create namespace outline
kubectl create secret tls shadowbox-tls -n outline --key
${SB_PRIVATE_KEY_FILE} --cert ${SB_CERTIFICATE_FILE}
```

3. Build a Kubernetes application for VPN key gener

For this step, you will need to acquire 2 regional static IP's f
Manager and Server LoadBalancers. Make sure to make note of the two IP addresses. We will be using IP's in this step and the Server LB static IP in the next step.

The Manager pod generates keys and stores them onto a persistent volume. We use Google disk to create this volume.

We will first create a deployment to act as an NFS provisioner on GKE using Google Disk. Run `kubectl apply -f` on the following manifest to create the provisioner (source):

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nfs-server
spec:
  replicas: 1
  selector:
    matchLabels:
      role: nfs-server
  template:
    metadata:
      labels:
        role: nfs-server
    spec:
      containers:
      - name: nfs-server
        image: gcr.io/google_containers/volume-nfs:0.8
        ports:
          - name: nfs
            containerPort: 2049
          - name: mountd
            containerPort: 20048
          - name: rpcbind
            containerPort: 111
        securityContext:
          privileged: true
        volumeMounts:
          - mountPath: /exports
            name: mypvc
      volumes:
```

```
        - name: mypvc
          gcePersistentDisk:
            pdName: gce-nfs-disk
            fsType: ext4
---
apiVersion: v1
kind: Service
metadata:
  name: nfs-server
spec:
  ports:
    - name: nfs
      port: 2049
    - name: mountd
      port: 20048
    - name: rpcbind
      port: 111
  selector:
    role: nfs-server
```

Once we have the NFS server deployed we can verify that it is running:

```
$ kubectl get pods
NAME                          READY     STATUS    RESTARTS   AGE
nfs-server-xxxxxxxxxx-xxxxx   1/1       Running   0          1m
```

To create our persistent volume, we need to obtain the ClusterIP used by the `nfs-server` service:

```
$ kubectl get svc
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
kubernetes   ClusterIP   xx.xx.xx.x     <none>        443/TCP    1d
nfs-server   ClusterIP   10.124.12.49   <none>        ...        1m
```

We will then replace this value into the following manifest (value marked in bold) and run `kubectl apply -f`:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs
  namespace: outline
spec:
```

```
  capacity:
    storage: 2Gi
  accessModes:
    - ReadWriteMany
  nfs:
    server: 10.124.12.49 #Replace with nfs-server ClusterIP
    path: "/"
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: outline-pvc
  namespace: outline
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: ""
  resources:
    requests:
      storage: 2Gi
```

In order to successfully pull the Docker image we created earlier, we need to provide the credentials required to pull from our private registry.

```
$ kubectl create secret docker-registry regcred \
      --docker-server=<your-registry-server> \
      --docker-username=<your-username> \
      --docker-password=<your-pword> \
      --docker-email=<your-email> \
      --namespace=outline
```

Now that we have the volume and the secrets required, we can go ahead and create the Manager pod. Run `kubectl apply -f` on the following manifest. Note that there are a few variables you will need to replace (value marked in bold):

```
apiVersion: v1
kind: Pod
metadata:
  namespace: outline
  name: shadowbox-manager
  labels:
    app: shadowbox-manager
spec:
  initContainers:
  - name: install
    image: busybox
```

```
      command: ["/bin/sh"]
      args: ["-c", "touch /tmp/outline-ss-server/config.yml; echo
'{\"rollouts\":[{\"id\":\"single-
port\",\"enabled\":true}],\"portForNewAccessKeys\":443}' >
/tmp/shadowbox_server_config.json; if [ ! -f
/tmp/shadowbox_config.json ]; then echo 'USER CONFIG NOT FOUND';
echo '{\"defaultPort\":443,\"accessKeys\":[],\"ne
/tmp/shadowbox_config.json; fi"]
      volumeMounts:
      - name: shadowbox-config
        mountPath: /tmp
    containers:
    - name: shadowbox-manager
      image: <registry>/<project path>:<version>
      env:
      - name: LOG_LEVEL
        value: "debug"
      - name: SB_API_PREFIX
        value: TestApiPrefix
      - name: SB_CERTIFICATE_FILE
        value: "/tmp/shadowbox-selfsigned-dev.crt"
      - name: SB_PRIVATE_KEY_FILE
        value: "/tmp/shadowbox-selfsigned-dev.key"
      - name: SB_API_PORT
        value: "8081"
      - name: SB_PUBLIC_IP
        value: "xx.xx.xx.xx" #Server LB static IP (GCP)
      volumeMounts:
      - name: shadowbox-config
        mountPath: /root/shadowbox/persisted-state
      - name: server-config-volume
        mountPath: /cache
      - name: tls
        mountPath: /tmp/shadowbox-selfsigned-dev.crt
        subPath: shadowbox-selfsigned-dev.crt
      - name: tls
        mountPath: /tmp/shadowbox-selfsigned-dev.key
        subPath: shadowbox-selfsigned-dev.key
      ports:
      - containerPort: 80
      - containerPort: 8081
    volumes:
    - name: server-config-volume
      emptyDir: {}
    - name: shadowbox-config
      persistentVolumeClaim:
        claimName: outline-pvc
    - name: tls
      secret:
        secretName: shadowbox-tls
        items:
        - key: tls.crt
          path: shadowbox-selfsigned-dev.crt
        - key: tls.key
          path: shadowbox-selfsigned-dev.key
```

```
  imagePullSecrets:
  - name: regcred
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: shadowbox-manager
  namespace: outline
  name: shadowbox-manager-tcp-service
spec:
  type: LoadBalancer
  loadBalancerIP: "yy.yy.yy.yy" #Manager LB static IP (GCP)
  ports:
  - name: out
    port: 8081
    targetPort: 8081
    protocol: TCP
  selector:
    app: shadowbox-manager
```

The two IP addresses that you will replace are `SB_PUBLIC_IP` (static IP address for your Server LoadBalancer) and `loadBalancerIP` (static IP for your Manager LoadBalancer).

Note: the `initContainer` runs a few commands to achieve the following:

- Use single-port settings for each key and set the port to 443
- Mount the key files (config.yml and shadowbox_server_config.json) onto the persistent volume

4. Build a Kubernetes application for hosting the VPN server

A similar configuration is used for the Server pods deployment. The main difference is the `initContainer` which is replaced by the `postStart` lifecycle hook in which we run commands to accomplish the following:

- Set server settings to single-port, using port 443
- Copying the two key files (config.yml and shadowbox_server_config.json) from Manager pod onto their location on the Server pods through the mounted volume
- A 10-second delay for the server to start functioning with the configured settings — this merely starts the pod with previously created keys, in order for the Server

pod to function with any keys added later on we need to perform the steps after

- Create a symlink for each of the two key files (config.yml and shadowbox_server_config.json) that points to the original files on the mounted volume
- Add a cronjob to restart the Server pods every 15 minutes, in order to adopt the new key configurations

The following deployment creates 3 Server replicas. Replace the values marked in bold and run `kubectl apply -f` on the manifest file:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: outline
  name: shadowbox-server
  labels:
    app: shadowbox
spec:
  replicas: 3
  selector:
    matchLabels:
      app: shadowbox
  template:
    metadata:
      labels:
        app: shadowbox
    spec:
      containers:
      - name: shadowbox
        image: <registry>/<project path>:<version>
        lifecycle:
          postStart:
            exec:
              command: ["/bin/sh", "-c", "echo '{\"rollouts\":
[{\"id\":\"single-
port\",\"enabled\":true}],\"portForNewAccessKeys\":443}' >
/root/shadowbox/persisted-state/shadowbox_server_config.json; cat
/opt/outline/shadowbox_config.json > /root/shadowbox/persisted-
state/shadowbox_config.json; cat /opt/outline/outline-ss-
server/config.yml > /root/shadowbox/persisted-state/outline-ss-
server/config.yml; sleep 10; ln -sf
/opt/outline/shadowbox_config.json /root/shadowbox/persisted-
state/shadowbox_config.json; ln -sf /opt/outline/outline-ss-
server/config.yml /root/shadowbox/persisted-state/outline-ss-
server/config.yml; var='kill -SIGHUP $(pgrep -f outline-ss-server)';
echo \"*/15 * * * * $var\" > mycron; crontab mycron; rm mycron;"]
        env:
```

```
        - name: LOG_LEVEL
          value: "debug"
        - name: SB_API_PREFIX
          value: TestApiPrefix
        - name: SB_CERTIFICATE_FILE
          value: "/tmp/shadowbox-selfsigned-dev.crt"
        - name: SB_PRIVATE_KEY_FILE
          value: "/tmp/shadowbox-selfsigned-dev.k
        - name: SB_PUBLIC_IP
          value: "xx.xx.xx.xx" #Server LB static IP (GCP)
        volumeMounts:
        - name: shadowbox-config
          mountPath: /opt/outline
          readOnly: true
        - name: server-config-volume
          mountPath: /cache
        - name: tls
          mountPath: /tmp/shadowbox-selfsigned-dev.crt
          subPath: shadowbox-selfsigned-dev.crt
        - name: tls
          mountPath: /tmp/shadowbox-selfsigned-dev.key
          subPath: shadowbox-selfsigned-dev.key
        ports:
        - containerPort: 80
        - containerPort: 443
      volumes:
      - name: server-config-volume
        emptyDir: {}
      - name: shadowbox-config
        persistentVolumeClaim:
          claimName: outline-pvc
      - name: tls
        secret:
          secretName: shadowbox-tls
          items:
          - key: tls.crt
            path: shadowbox-selfsigned-dev.crt
          - key: tls.key
            path: shadowbox-selfsigned-dev.key
      imagePullSecrets:
      - name: regcred
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: shadowbox
  namespace: outline
  name: shadowbox-lb-tcp
spec:
  type: LoadBalancer
  loadBalancerIP: "xx.xx.xx.xx" #Server LB static IP (GCP)
  ports:
  - name: out
    port: 443
```

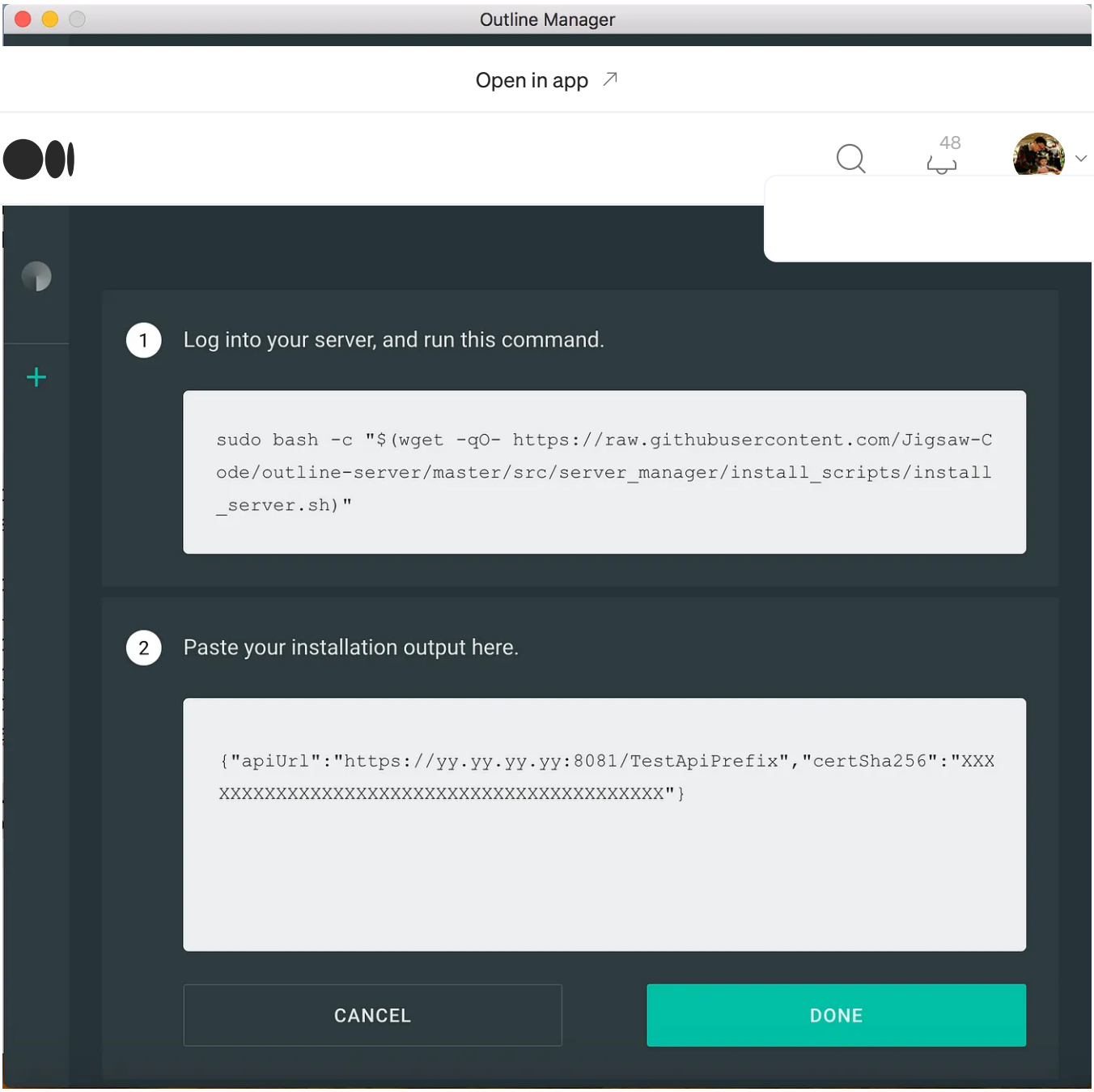
```
    targetPort: 443
    protocol: TCP
  selector:
    app: shadowbox
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: shadowbox
  namespace: outline
  name: shadowbox-lb-udp
spec:
  type: LoadBalancer
  loadBalancerIP: "zz.zz.zz.zz" #Server LB static IP (GCP)
  ports:
  - name: out
    port: 443
    targetPort: 443
    protocol: UDP
  selector:
    app: shadowbox
```

5. Output associated encrypted string to use in Outline Manager

Your service is now ready to create keys through the Outline Manager using the Manager pod and serve as a VPN service using the Server pods. You will need to run the following command to obtain the string needed to insert into Outline Manager:

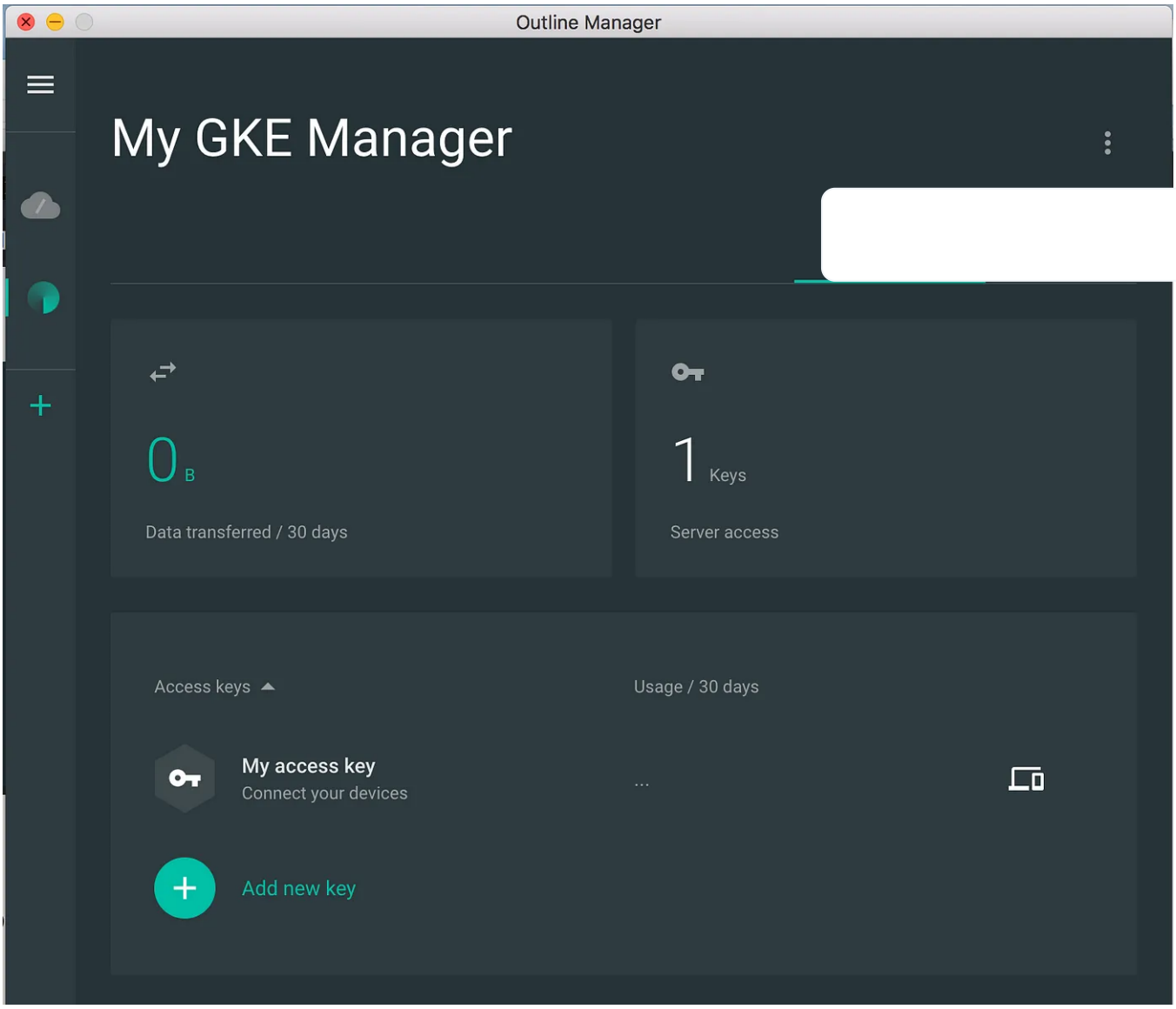
```
$ export SHA=$(openssl x509 -noout -fingerprint \
    -sha256 -inform pem -in ${SB_CERTIFICATE_FILE} \
    | sed "s/://g" | sed 's/.*=//')

$ export MANAGER_LB_ADDRESS=$(kubectl get svc -n \
    outline shadowbox-manager-tcp-service \
    -o jsonpath="{.status.loadBalancer.ingress[0].hostname}")

$ echo \
{\"apiUrl\":\"https://${MANAGER_LB_ADDRESS}:8081/TestApiPrefix\",\"c
ertSha256\":\"${SHA}\"\}

{"apiUrl":"https://yy.yy.yy.yy:8081/TestApiPrefix","certSha256":"XXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"}
```

You can then copy the output string and use inside Outline Manager:



Outline Manager

Click “DONE”, and now you can create keys by clicking the “Add new key” button at the bottom of the window:



You can now connect any computer to your VPN server using your created keys.

Note: when you create a key, there is a wait time of up to 15 minutes until the key is activated. This is done by the cronjob you set up earlier in your Server pods' `PostStart` LifeCycle hook which restarts the service every 15 minutes.

Enjoy!

Kubernetes VPN Outline Shadowsocks High Availability

Written by Lexa

70 Followers · Writer for ASL19 Developers

DevOps and Backend at @asl19-engineering

More from Lexa and ASL19 Developers

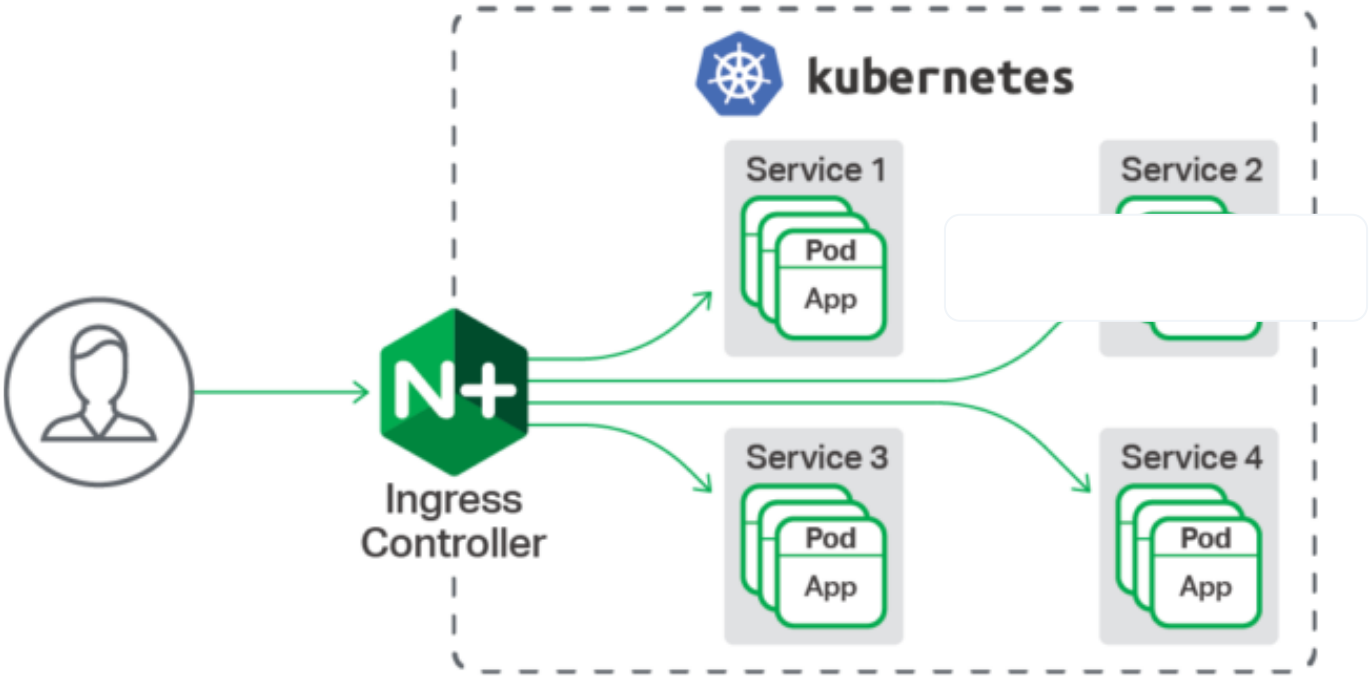
Lexa in ASL19 Developers

Create ReadWriteMany PersistentVolumeClaims on your Kubernetes Cluster

Kubernetes allows us to provision our PersistentVolumes dynamically using PersistentVolumeClaims. Pods treat these claims as volumes. The...

· 8 min read · May 17, 2019

248 8



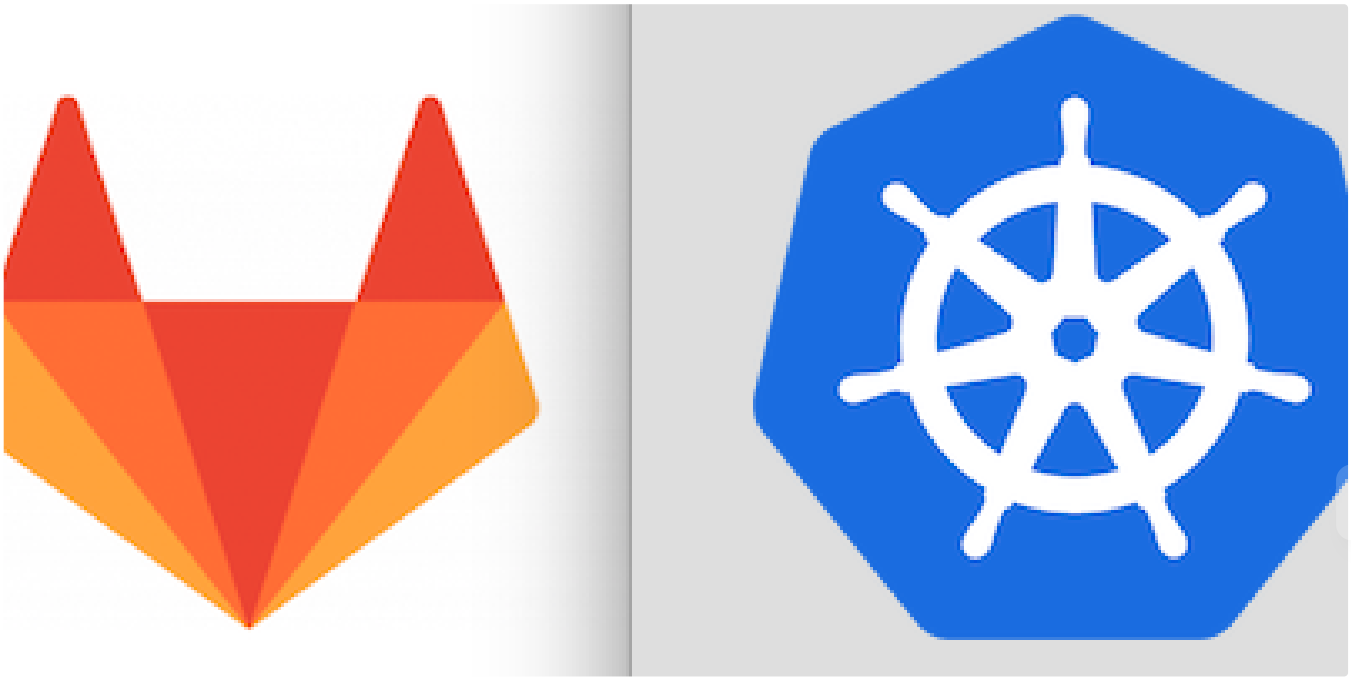
Lexa in ASL19 Developers

Use Let's Encrypt, Cert-Manager and External-DNS to publish your Kubernetes apps to your website

In this tutorial, we aim to bring a few tools together to automate the process of publishing your Kubernetes applications to your website...

11 min read · May 17, 2019

88 1



Lexa in ASL19 Developers

Connect your Kubernetes cluster to multiple GitLab projects for CI/CD

My team has recently boarded the hype train on Kubernetes deployments. We can get rid of a bunch of staging servers and never look back...

3 min read · May 17, 2019

69 2

Lexa in ASL19 Developers

Build your own cloud-agnostic TCP/UDP LoadBalancer for your Kubernetes apps

Recently I've been working on a Kubernetes migration for a project with a specific requirement for TCP/UDP load-balancing. I deployed this...

3 min read · May 17, 2019

61 1

See all from Lexa

See all from ASL19 Developers

Recommended from Medium

headinthecloudsin Dev Genius

2 Methods: Deploy an App with a Basic Ingress Service on AWS EKS Using Kubernetes and Terraform...

· 9 min read · Mar 6

7

Ahmed Elfakharany

Helm vs Kustomize: why, when, and how

The challenge

· 12 min read · Jun 10

Lists

Now in AI: Handpicked by Better Programming

248 stories · 11 saves

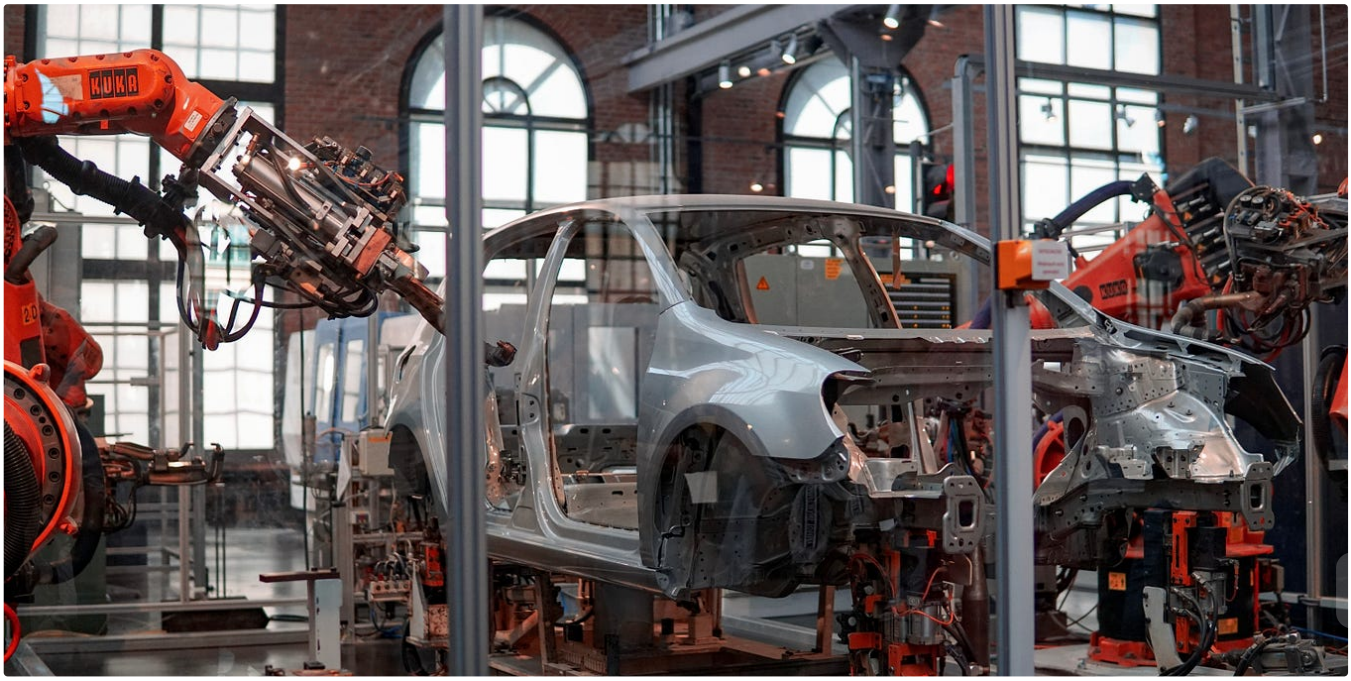
StringMeteor in Level Up Coding

Run a Kubernetes cluster on Apple Silicon Mac with kind

How to set up a Kubernetes cluster inside a Docker container on a Mac in just a few commands

· 9 min read · Jan 30

86



Kevin Summersill in Geek Culture

Deploy a Virtual Machine to Proxmox with Terraform

Hashicorp | Virtualization | Automation | DevOps | Ubuntu Lunar |

✦ · 6 min read · Dec 28, 2022

12 1

StringMeteor in Level Up Coding

Docker on Apple Silicon Mac: How to Run x86 Containers with Rosetta 2

Running x86 containers on Apple Silicon Macs just got easier thanks to newly added Docker's Rosetta support. Discover how to use it in this...

✦ · 5 min read · Jan 16

176 9

Magsther

JupyterHub on Kubernetes

An introduction to JupyterHub

6 min read · Dec 30, 2022

18

See more recommendations