Kubernetes Cluster Deployment

We use the Google Cloud Platform as an environment and Argo CD as a tool to deploy our application on the cloud. Argo CD is a declarative, GitOps continuous delivery tool for Kubernetes. Argo CD automates the deployment of the desired application states in the specified target environments. Application deployments can track updates to branches, tags, or pinned to a specific version of manifests at a Git commit. Initially, we created a new cluster in the Google Kubernetes Engine.

- Argo CD is required to install to the Kubernetes cluster in order to be runnable, so we have created a namespace “argocd” by using;

kubectl create namespace argocd

- Argo CD services and application resources will live in this namespace by using;

kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml

- By default, the Argo CD API server is not exposed with an external IP. To access the API server, the Argo CD API server needs to be exposed. The argocd-server service type is changed to LoadBalancer by using :

kubectl patch svc argocd-server -n argocd -p '{"spec": {"type": "LoadBalancer"}}'

- Argo CD GUI endpoint can be found at Services & Ingress in Kubernetes Engine. This endpoint will you take to the login page for Argo CD. The initial password for the admin account is auto-generated and stored as clear text in the field password in a secret named argocd-initial-admin-secret in your Argo CD installation namespace. You can simply retrieve this password using kubectl:

kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d; echo

- The application could be added to the Argo CD either using the Argo CD dashboard or with a .yaml configuration file using:

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: {application name}
  namespace: argocd
spec:
  project: default

  source:
    repoURL: {git repo url}
    targetRevision: HEAD
    path: {path contains kubernetes manifest files}
  destination:
    server: https://kubernetes.default.svc
    namespace: {namespace to application}

  syncPolicy:
    syncOptions:
    - CreateNamespace=true

    automated:
      selfHeal: true
      prune: true
```

A more detailed explanation can be found at the given link https://argo-cd.readthedocs.io/en/stable/getting_started/

SSL/TLS Certification

In Kubernetes, SSL certificates are stored as Kubernetes secrets. Certificates are usually valid for one to two years after which they expire so there's a big management overhead and potential for some downtime. We'll want a setup that is self-managed and automatically renews certificates that expire.

This is where Cert-manager comes in. Cert-manager is a resource we deploy in our cluster that can talk to certificate authorities like Let's Encrypt (which is free) to generate certificates for our domain. Initially, we deployed cert-manager in our cluster;

curl -LO https://github.com/jetstack/cert-manager/releases/download/v1.8.0/cert-manager.yaml

Then let's deploy cert-manager to a namespace called cert-manager;

```
kubectl create namespace cert-manager

kubectl apply --validate=false -f cert-manager.yaml
```

In order to hook up cert-manager to a certificate authority like Let's Encrypt another Kubernetes object called an Issuer needs to be deployed. We set up the Issuer that specifies the server for certificate authority and the name of Kubernetes secret key reference where the issuer key should be stored.

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-cluster-issuer
spec:
  acme:
    server: https://acme-v02.api.letsencrypt.org/directory
    email: your-email@email.com
    privateKeySecretRef:
      name: letsencrypt-cluster-issuer-key
    solvers:
    - http01:
        ingress:
          class: nginx
```

Let's deploy the Issuer.

```
kubectl apply -f cert-issuer.yaml# view the

kubectl describe clusterissuer letsencrypt-cluster-issuer
```

We got the cert-manager and the issuer in place. Now we can request a certificate. Here's the template for the certificate object.

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: example-cert #name of this object
  namespace: default #same namespace as
spec:
  dnsNames:
    - example.com
  secretName: example-tls-cert
  issuerRef:
    name: letsencrypt-cluster-issuer
    kind: ClusterIssuer
```

In the template we specify the DNS name we want a certificate for, a secret name in Kubernetes secrets where the certificate should be stored, and a reference to the Issuer we deployed earlier. Also make sure to use the same name space where you'll deploy the service that will use this certificate. Let's deploy it;

```
kubectl apply -f certificate.yaml
```

We deployed the application and its service and we want to expose the application to the internet via an nginx ingress and set up TLS for it using the certificate that we issued above. Here is the template we used:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
    name: my-ingress
    annotations:
      kubernetes.io/ingress.class: nginx
spec:
  tls:
  - hosts:
    - example.com
  secretName: example-tls-cert

  rules:
  - host: example.com
    http:
      paths:
        - path: /
          pathType: Exact
          backend:
            service:
              name: backend-service
              port:
                number: 80
```

Under the tls section, we specify the DNS host for this ingress route and the secret name for the certificate we created earlier. We also pass the name of the service the ingress will route to.